

Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania



Anno Accademico 2026/2027

Indice

1	Introduzione alla Programmazione in C	4
1.1	Dalla Fisica all'Astrazione: Il Ruolo del Programmatore Oggi	4
1.1.1	Definizione di Programmazione, Rilevanza e Applicazioni	4
1.1.2	L'Epoca dell'IA ed i Dati dell'Industria	4
1.1.3	Ma allora, perché imparare a programmare nell'era dell'IA?	6
1.1.4	L'Informatica è la Nuova Fisica (Nobel 2024)	6
1.1.5	La Metafora del Direttore d'Orchestra (<i>Agentic Engineering</i>)	7
1.2	L'Evoluzione del Calcolo e l'Architettura di Von Neumann	7
1.2.1	La Pascalina e l'Informatica Meccanica (1642)	7
1.2.2	ENIAC e la Fisicità del Ricablaggio (Anni '40)	7
1.2.3	John von Neumann e lo <i>Stored Program Concept</i> (1945)	8
1.2.4	La Macchina Universale (CPU vs RAM)	8
1.3	I Livelli di Astrazione: Dal Silicio al C	8
1.3.1	Dal Codice Macchina al Linguaggio C	10
1.3.2	La Metafora dell'Automobile e del Meccanico	10
1.4	Sotto il Cofano: Registri e Traduzione Assembly	11
1.4.1	Il Collo di Bottiglia di Von Neumann ed i Registri CPU	11
1.4.2	Dal Codice Umano all'Assembly: L'Illusione delle Variabili	11
1.5	La Riga di Comando: Parlare Direttamente al Sistema Operativo	13
1.5.1	Cos'è e a cosa serve	13
1.5.2	Le Nostre Piattaforme: Shell Unix	14
1.5.3	Comandi e Scorciatoie di Base	14
1.6	Pratica al Terminale: La Magia del Valore di Ritorno (<code>echo \$?</code>)	15
1.6.1	Prima Compilazione e il Valore di Ritorno	15
1.6.2	Interrogare il Sistema Operativo con <code>echo \$?</code>	15
1.7	Dai Traduttori all'Hello World	16
1.7.1	Compilatori vs Interpreti	16
1.7.2	Anatomia della Toolchain del Compilatore (Le 4 Fasi)	16

1.7.3	Il Problema del Mondo Esterno (<code>stdio.h</code>)	23
1.8	Anatomia dell'Hello World e i Commenti nel Codice	24
1.8.1	Anatomia del Primo Programma C Reale	24
1.8.2	Sintassi dei Commenti in C	24
1.8.3	Il Punto e Virgola (;) come Terminatore	24
1.8.4	Whitespace e Formattazione Libera (Free-form)	25
1.8.5	L'Importanza dell'Indentazione	25
1.8.6	Verifica dell'Hello World con <code>echo \$?</code>	25
1.9	Quiz di Autovalutazione	26
1.10	Esercizi Pratici	29
2	La Logica dei Numeri e l'Input/Output	36
2.1	Variabili, Tipi di Dato e I/O Standard	36
2.1.1	Che cos'è una Variable sotto il cofano?	36
2.1.2	Il Concetto di Tipo	37
2.1.3	Interi vs Reali: <code>int</code> e <code>double</code>	37
2.1.4	Dichiarazioni con Inizializzazione e Dichiarazioni Disgiunte	37
2.1.5	Identità di Assegnamento: Lvalue vs Rvalue	38
2.2	Input/Output e Formattazione	38
2.2.1	La Filosofia di <code>printf</code>	38
2.2.2	Specificatori di Formato Essenziali	38
2.2.3	L'Input dall'Utente: La Funzione <code>scanf</code>	39
2.2.4	Esempio Pratico Completo: Calcolo dell'Area del Cerchio	39
2.3	Espressioni, Operatori e Logica	40
2.3.1	Espressioni Aritmetiche, Precedenza ed Associatività	40
2.3.2	Ogni espressione restituisce un valore!	40
2.3.3	Esempio di Valutazione delle Espressioni	41
2.3.4	Assegnamenti Composti	41
2.3.5	Il Trabocchetto della Divisione: Intera vs Reale	42
2.4	Logica Booleana e Decisioni Condizionali	42
2.4.1	Come il C rappresenta la Verità	42
2.4.2	Operatori Logici Booleani	43
2.4.3	L'Istruzione Condizionale <code>if</code>	44
2.4.4	La Logica Cortocircuitata (Short-Circuit Evaluation)	45
2.5	Quiz di Autovalutazione	45
2.6	Esercizi Pratici	50

3	La Cassetta degli Attrezzi del C	53
3.1	Il Sistema dei Tipi: Modificatori e Limiti	53
3.1.1	Riepilogo Rapido	53
3.1.2	Modificatori di tipo, il tipo <code>char</code> (ASCII) e la dipendenza dalla macchina	53
3.1.3	Scoprire le Dimensioni con <code>sizeof</code>	55
3.1.4	Accedere ai Limiti Esatti: <code><limits.h></code> e <code><float.h></code>	56
3.2	Codifica IEEE 754: Perché $0.1 + 0.2 \neq 0.3$	56
3.2.1	Perché esiste il bias 1023?	57
3.2.2	Esempio: rappresentazione di 0.1	57
3.2.3	Tabella dei campi del <code>double</code> per 0.1	57
3.2.4	Conseguenze Pratiche	58
3.3	Formati di Stampa <code>printf</code> — La Guida Completa	59
3.3.1	Tutti i Formati Essenziali	59
3.3.2	Controllo della Precisione Decimale	59
3.3.3	Ampiezza Minima e Padding	60
3.4	Costanti: Tre Modi per Dare un Nome a un Valore Fisso	60
3.4.1	Letterali Tipizzati	61
3.4.2	<code>const</code> — Variabile Immutabile	61
3.4.3	<code>#define</code> — Sostituzione del Preprocessore	61
3.4.4	Costanti Predefinite dalla Libreria Standard	62
3.5	Conversioni di Tipo: Implicite ed Esplicite	62
3.5.1	Conversione Implicita (Promozione)	62
3.5.2	Cast Esplicito	63
3.5.3	Arrotondamento con <code><math.h></code> : <code>round</code> , <code>floor</code> , <code>ceil</code>	63
3.6	La Libreria <code><math.h></code>	64
3.6.1	Perché una Libreria Separata?	64
3.6.2	Le Funzioni Principali	64
3.6.3	Demo Pratica: Calcolatore di Moto Uniformemente Accelerato	64
3.7	Overflow, Underflow, <code>inf</code> e <code>NAN</code>	65
3.7.1	Cosa Succede Quando Superiamo i Limiti degli Interi?	65
3.7.2	Overflow e Valori Speciali nei Numeri in Virgola Mobile	66
3.7.3	La Regola d'Oro: Compilate Sempre con <code>-Wall -Wextra</code>	66
3.8	Quiz di Autovalutazione	66
3.9	Esercizi Pratici	71

4	Strutture Decisionali e l'Iterazione Fondamentale	82
4.1	Dalla RAM alla Programmazione Strutturata (Sotto il Cofano)	82
4.1.1	Il Flusso non Lineare e il Trucco del Salto (Goto)	82
4.1.2	La Flessibilità del goto e lo Spaghetti Code	83
4.1.3	Il Teorema di Böhm-Jacopini	83
4.1.4	Concettualizzare con i Diagrammi di Flusso (Flowchart)	84
4.2	Le Strutture Condizionali in C	84
4.2.1	La Selezione Completa: if, else ed else if	84
4.2.2	Selezione Multipla Discreta: switch	85
4.2.3	L'Operatore Ternario (? :)	86
4.3	L'Iterazione Fondamentale: Il Ciclo while	86
4.3.1	Il Costrutto while	87
4.3.2	Evitare il Loop Infinito	87
4.3.3	Algoritmo del Massimo e Minimo (Sequenza Fissa)	87
4.3.4	Modificare il Flusso del Ciclo: break e continue	89
4.4	Quiz di Autovalutazione	91
4.5	Esercizi Pratici	94
5	Iterazione Avanzata, Visibilità, Casualità e Debugging	105
5.1	Iterazione Avanzata: I Cicli for e do-while	105
5.1.1	Il Ciclo for (Iterazione controllata da contatore)	105
5.1.2	Il Ciclo do-while (Iterazione post-condizionata)	106
5.2	Cicli Annidati e Visibilità delle Variabili (Scope & Lifetime)	107
5.2.1	Cicli Annidati (Nested Loops)	108
5.2.2	Scope e Lifetime delle Variabili	108
5.3	Numeri Pseudo-Casuali e Debugging	109
5.3.1	Generazione di Numeri Pseudo-Casuali	110
5.3.2	La Cultura del Debugging: LLDB da Terminale e VS Code	111
5.4	Quiz di Autovalutazione	113
5.5	Esercizi Pratici	118
6	Array Monodimensionali Statici e Algoritmi Base	129
6.1	Motivazione e Anatomia degli Array	129
6.1.1	Il Problema che Motiva l'Array	129
6.1.2	Dichiarazione e Layout in RAM	129
6.1.3	Inizializzazione e Uso di #define N	130
6.1.4	La dimensione di un Array: L'operatore sizeof	132

6.2	Il Pattern di Scansione e gli Algoritmi Base	132
6.2.1	Il Ciclo <code>for</code> come Strumento Universale sugli Array	132
6.2.2	Algoritmo: Calcolo della Media	133
6.2.3	La Prima Vera Dimostrazione del Valore degli Array: la Seconda Passata	134
6.3	Algoritmi di Ricerca e di Estremi	134
6.3.1	Algoritmo: Ricerca Lineare (Sequential Search)	134
6.3.2	Algoritmo: Massimo e Minimo su Array	135
6.3.3	Esempio Integrato: Analisi Statistica dei Voti	136
6.4	Quiz di Autovalutazione	137
6.5	Esercizi Pratici	141
7	L'Astrazione del Codice e l'Anatomia dello Stack	153
7.1	Il Limite del Codice Monolitico e il Principio DRY	153
7.1.1	Perché il copia-incolla è il male	153
7.1.2	L'approccio procedurale	154
7.2	Anatomia di una Funzione in C	155
7.2.1	Prototipi e Definizioni	156
7.3	Il Flusso di Esecuzione e lo Scope delle Variabili	156
7.3.1	Alterazione del flusso	157
7.3.2	Passaggio per Valore (Copia)	158
7.3.3	Scope Locale (Block Scope)	159
7.4	Sotto il Cofano: Stack e Record di Attivazione	160
7.4.1	Organizzazione della RAM	160
7.4.2	Lo Stack Frame (Record di Attivazione)	161
7.4.3	Allineamento e Padding a 16 Byte	161
7.4.4	I Registri RSP e RBP	162
7.5	Quiz di Auto-valutazione	163
7.6	Esercizi Pratici	168
8	I Puntatori e la Gestione Esplicita della Memoria	177
8.1	La Realtà della RAM e i Tipi Puntatore	177
8.1.1	Oltre l'illusione delle variabili	177
8.1.2	Dichiarazione e interpretazione dei puntatori	178
8.1.3	Operatori <code>&</code> (Indirizzo) e <code>*</code> (Dereferenziazione)	180
8.2	Passaggio per Riferimento e la funzione <code>Swap</code>	183
8.2.1	Il limite del passaggio per valore	183
8.2.2	Il passaggio per riferimento simulato	184

8.2.3	La corrispondenza tra Array e Puntatori	184
8.2.4	Il decadimento degli array (Array Decay)	185
8.2.5	Aritmetica dei puntatori e zucchero sintattico	186
8.2.6	Sicurezza: Il modificatore const	187
8.2.7	Funzioni inline	188
8.3	Quiz di Auto-valutazione	190
8.4	Esercizi Pratici	194
9	Le Stringhe e lo Spazio Multidimensionale	204
9.1	L'Anatomia delle Stringhe e il Terminatore Null	204
9.1.1	L'Astrazione del Testo (Il tipo che non esiste)	204
9.1.2	Inizializzazione e l'Importanza del Terminatore <code>\0</code>	204
9.1.3	Lunghezza Massima vs Lunghezza Logica	205
9.1.4	Scansione di Stringhe ed Aritmetica dei Puntatori	206
9.2	Manipolazione, Librerie Standard e Sicurezza	207
9.2.1	Analisi dei Caratteri con <code><ctype.h></code>	207
9.2.2	Conversione Sicura di Stringhe (<code>strtod</code> vs <code>atoi</code>)	207
9.2.3	Uso della Libreria <code><string.h></code>	208
9.3	L'Input/Output di Testo Sicuro e le Matrici	209
9.3.1	La piaga del Buffer Overflow e la soluzione <code>fgets</code>	209
9.3.2	Le Matrici (Linearizzazione in Row-Major Order)	210
9.3.3	Matrici di Caratteri come Array di Stringhe	211
9.3.4	Array N-Dimensionali	213
10	Algoritmi Iterativi di Ordinamento e Ricerca	215
10.1	Il Ruolo dell'Ordinamento e lo Swap	215
10.1.1	Perché Ordiniamo i Dati?	215
10.1.2	La Funzione <code>swap</code> con Puntatori	215
10.2	Il Mescolamento Casuale: Fisher-Yates Shuffle	216
10.2.1	Perché lo Shuffling "Ingenuo" è Sbagliato	216
10.2.2	L'Algoritmo di Fisher-Yates	216
10.3	I Tre Algoritmi di Ordinamento Iterativi	217
10.3.1	Selection Sort: "Trova il Minimo"	218
10.3.2	Bubble Sort: "Le Bollicine in RAM"	219
10.3.3	Insertion Sort: "L'Ordinamento delle Carte"	221
10.4	La Ricerca Dicotomica (Binary Search)	222
10.4.1	Ricerca Lineare vs Dicotomica	222

10.4.2	Prevenire l'Overflow nel Calcolo del Punto Medio	223
10.4.3	Analisi quantitativa dell'efficienza	224
10.5	Quiz di Auto-valutazione	225
10.6	Esercizi Pratici	229
11	Interazione con l'OS: CLI, Flussi di Testo e Tokenizzazione	242
11.1	La Riga di Comando e lo Standard Error (CLI & STDERR)	242
11.1.1	Il Limite dell'Isolamento interattivo	242
11.1.2	L'Anatomia di <code>argc</code> e <code>argv</code> nello Stack	242
11.1.3	Lo Standard Error (<code>stderr</code>)	244
11.2	I Limiti dell'Input e l'Acquisizione Robusta	245
11.2.1	Le insidie nascoste di <code>scanf</code>	245
11.2.2	Tecniche di Lettura Sicura	245
11.2.3	EOF, <code>getchar</code> e la lettura carattere per carattere	246
11.3	Persistenza, File Stream e Tokenizzazione in Memoria	247
11.3.1	Apertura e Chiusura dei File: il tipo <code>FILE</code>	247
11.3.2	Il Pericolo di <code>fscanf</code> nei File Reali	248
11.3.3	Robust Line-by-Line Parsing	248
11.3.4	Strumenti di Tokenizzazione: <code>sscanf</code> e <code>strtok</code>	248
11.4	Quiz di Auto-valutazione	250
11.5	Esercizi Pratici	254
12	Basso Livello, File Binari e l'Abisso della Ricorsione	264
12.1	I File Binari e l'Ispezione a Basso Livello	264
12.1.1	Testo contro Binario: L'Illusione ASCII	264
12.1.2	La Scrittura Binaria con <code>fwrite</code>	265
12.1.3	Hexdump e la Rappresentazione Little Endian	265
12.2	Accesso Casuale e Gestione dei Record Binari	266
12.2.1	Salvataggio di Array in Blocco	266
12.2.2	Navigazione nel File con <code>fseek</code> e <code>ftell</code>	266
12.2.3	Record a Dimensione Fissa e Stringhe Variabili	267
12.3	L'Abisso della Ricorsione e i Limiti dello Stack	268
12.3.1	I Principi della Ricorsione	268
12.3.2	Esempi Algoritmici Classici	268
12.3.3	Lo Stack Frame e la Catastrofe dello Stack Overflow	270
12.3.4	Inefficienza e la Ricorsione di Coda (Tail Recursion)	270
12.4	Quiz di Auto-valutazione	272
12.5	Esercizi Pratici	276

13 Modellare la Realtà (Struct, Padding e Persistenza a Blocchi)	293
13.1 La Fine degli Array Paralleli e l'Incapsulamento	293
13.1.1 Il Problema: I limiti dei tipi predefiniti	293
13.1.2 La Creazione di un Nuovo Tipo: La struct	294
13.1.3 Inizializzazione e Assegnazione Magica	295
13.2 Puntatori a Struct e Fisica della Memoria	296
13.2.1 Passaggio per Valore vs Riferimento: L'Operatore Freccia (->)	296
13.2.2 Sotto il Cofano: Padding e Allineamento (Data Alignment)	297
13.2.3 Enumerazioni (enum): Un tocco di leggibilità	299
13.3 Il Trionfo di fread, fwrite ed fseek	300
13.3.1 Unire la Teoria: Struct e File Binari	300
13.3.2 Il Salvataggio Istantaneo dell'intero Database	300
13.3.3 Accesso Diretto	301
13.4 Quiz di Auto-valutazione	302
13.5 Esercizi Pratici	306
 14 La Fuga dallo Stack (Memoria Dinamica e la nascita del Nodo)	 321
14.1 Il Muro dello Stack e l'Ingresso nello Heap	321
14.1.1 Il problema irrisolvibile: Variabili Locali e Dangling Pointer	321
14.1.2 La Fuga: La Memoria HEAP	322
14.1.3 Le Chiavi dello Heap: malloc e free	323
14.2 Array Dinamici e Ridimensionamento	325
14.2.1 Sintassi di Accesso negli Array Dinamici	325
14.2.2 calloc: Allocazione con Inizializzazione a Zero	325
14.2.3 Ridimensionare la memoria: realloc	326
14.3 Verso le Strutture Dinamiche: Il Nodo	327
14.3.1 Superare il vincolo della contiguità	327
14.3.2 Il Nodo Autoreferenziale	327
14.3.3 Collegamento manuale di nodi	328
14.4 Autovalutazione ed Esercizi	328
14.4.1 Test a Risposta Multipla (10 Quiz)	329
14.4.2 Esercizi di Programmazione Progressivi	333

15 Liste Concatenate Semplici	338
15.1 La Struttura Logica della Lista	338
15.1.1 L'Anatomia di un Nodo	338
15.2 Rappresentazione in C ed Ispezione	339
15.2.1 Funzione di Allocazione Sicura	339
15.2.2 Scorrimento Sequenziale e Stampa	340
15.2.3 Ricerca di un Elemento	340
15.3 La Chirurgia dei Puntatori: Inserimenti	341
15.3.1 Inserimento in Testa	341
15.3.2 Inserimento in Coda	342
15.3.3 Inserimento Ordinato	343
15.4 La Chirurgia dei Puntatori: Cancellazioni	344
15.4.1 Cancellazione in Testa	345
15.4.2 Cancellazione in Coda	345
15.4.3 Cancellazione di un Elemento Specifico	346
15.4.4 Deallocazione dell'Intera Lista	348
15.4.5 Ordinamento su Liste (Bubble Sort)	348
15.5 Quiz di Autovalutazione	349
15.6 Esercizi Pratici Risolti	354
 16 Pile, Code e Liste Doppie	 365
16.1 Il Concetto di Tipo di Dato Astratto (ADT)	365
16.2 La Pila (Stack)	365
16.2.1 Metafore e Casi d'Uso	366
16.2.2 Implementazione di una Pila in C	366
16.3 La Coda (Queue)	368
16.3.1 Metafore e Casi d'Uso	368
16.3.2 Testa, Coda ed Efficienza	369
16.3.3 Implementazione di una Coda in C	369
16.4 Liste Doppiaemente Concatenate	371
16.4.1 La Struttura dei Nodi Doppi	372
16.4.2 Inserimenti Bilaterali	373
16.4.3 Rimozione Istantanea	373
16.4.4 I Trade-off Ingegneristici	374
16.5 Quiz di Autovalutazione	374
16.6 Esercizi Pratici Risolti	378

A Guida al Setup dell'Ambiente	387
A.1 Installazione della Toolchain (Compilatore e Debugger)	387
A.1.1 Piattaforma Windows (via WSL)	387
A.1.2 Piattaforma macOS	388
A.1.3 Piattaforma Linux (Ubuntu/Debian)	389
A.2 Installazione e Configurazione di Visual Studio Code	390
A.2.1 Passo 1: Download di VS Code	390
A.2.2 Passo 2: Importazione del Profilo del Corso (Prog1-0Z)	390
A.3 Test di Compilazione ed Esecuzione da Terminale	390
A.3.1 Passo 1: Scrivere il codice sorgente	391
A.3.2 Passo 2: Compilare con GCC o Clang	391
A.3.3 Passo 3: Eseguire il programma	391
A.4 Primi Passi con il Debugger lldb	391
A.4.1 Compilazione per il Debugging (Flag -g)	391
A.4.2 Comandi Fondamentali di lldb	392
A.5 Integrazione del Debugger in VS Code (Estensione CodeLLDB)	392
A.5.1 Passo 1: Verificare CodeLLDB in VS Code	393
A.5.2 Passo 2: Configurazione Progetto per Progetto	393
A.5.3 Passo 3: Come usare il Debug Grafico	394

LEZIONE 1

Introduzione alla Programmazione in C



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 1

Introduzione alla Programmazione in C

Benvenuti a **Programmazione 1**! Il nostro obiettivo in questa lezione è demistificare la “scatola nera” del computer, comprendere l’evoluzione della programmazione nell’era dell’Intelligenza Artificiale, collegare l’hardware ai concetti di astrazione del C e muovere i primi passi pratici al terminale con la compilazione ed l’ispezione dei valori di ritorno (echo \$?).

1.1 Dalla Fisica all’Astrazione: Il Ruolo del Programmatore Oggi

1.1.1 Definizione di Programmazione, Rilevanza e Applicazioni

La **programmazione** non consiste nel memorizzare sintassi o comandi a memoria. È l’arte di modellare il **pensiero algoritmico**: la capacità di prendere problemi complessi, disordinati e reali della scienza e della società e tradurli in strutture logiche trasparenti ed istruzioni deterministiche comprese dalle macchine.

La programmazione è oggi il motore invisibile di ogni disciplina moderna: dalle simulazioni dei cambiamenti climatici alla bioinformatica avanzata, dai sistemi aerospaziali alla medicina di precisione, fino alla domotica ed alla finanza.

1.1.2 L’Epoca dell’IA ed i Dati dell’Industria

Iniziamo il nostro percorso affrontando apertamente il contesto tecnologico in cui ci troviamo. L’Intelligenza Artificiale Generativa ha trasformato radicalmente il panorama del software engineering.

Analizziamo i dati reali ed ufficiali dell’industria:

- **GitHub Octoverse (2023)**: Il **92%** degli sviluppatori usa o sperimenta strumenti di coding basati su AI, anche fuori dal lavoro.
- **World Economic Forum (Future of Jobs Report 2023)**: Basato su 803 aziende e oltre 11,3 milioni di lavoratori, il report stima che il **44%** delle competenze sarà trasformato nei prossimi cinque anni.
- **Gartner (2024)**: Entro il 2028, il **75%** dei programmatori in ambito enterprise userà assistenti AI per il codice.



Figura 1.1: La Pervasività della Programmazione

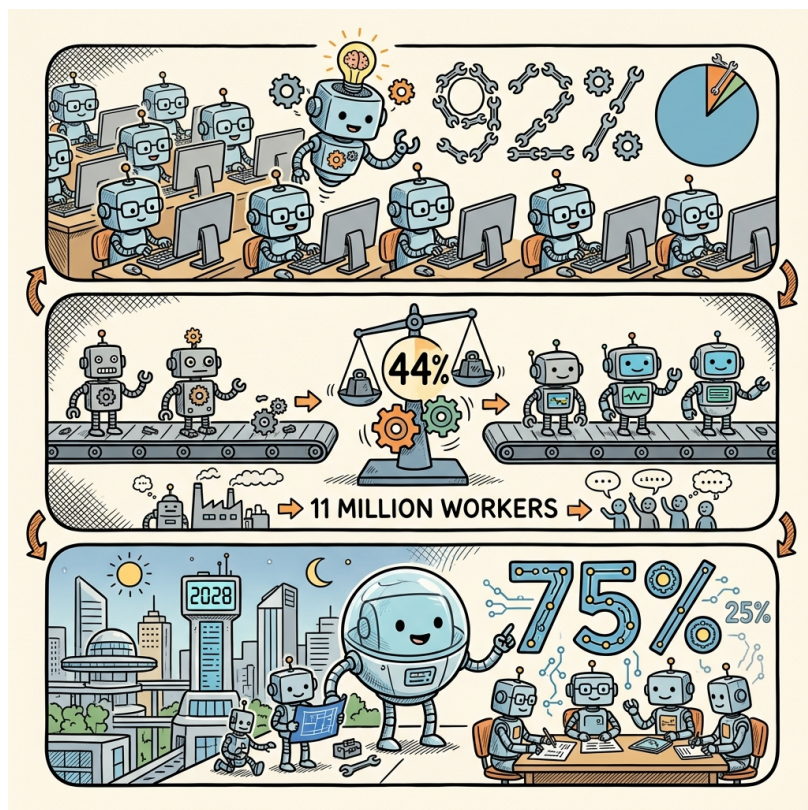


Figura 1.2: Infografica Dati IA



Figura 1.3: Premi Nobel 2024

1.1.3 Ma allora, perché imparare a programmare nell'era dell'IA?

Se gli strumenti IA possono generare codice in pochi millisecondi, qual è il valore dello studio dell'informatica?

L'Intelligenza Artificiale non è magia: è costruita interamente su strutture dati, allocazione di memoria, matrici, algoritmi ed equazioni algebriche. Imparare a programmare significa padroneggiare la grammatica ed la sintassi fondamentale di questo linguaggio per guidare la rivoluzione tecnologica anziché subirla.

1.1.4 L'Informatica è la Nuova Fisica (Nobel 2024)

L'informatica è diventata il nuovo telescopio e microscopio della scienza moderna. A conferma di ciò, i **Premi Nobel 2024** hanno consacrato il ruolo centrale degli algoritmi:

- **Nobel per la Fisica:** Assegnato a John Hopfield e Geoffrey Hinton per le scoperte fondamentali che consentono l'apprendimento automatico tramite reti neurali artificiali basate sulla fisica statistica.
- **Nobel per la Chimica:** Assegnato a Demis Hassabis, John Jumper e David Baker per la progettazione computazionale delle proteine e per **AlphaFold**, l'algoritmo che ha risolto il problema cinquantennale del ripiegamento proteico.



Figura 1.4: La Metafora del Direttore d'Orchestra

1.1.5 La Metafora del Direttore d'Orchestra (Agentic Engineering)

L'IA non sostituisce il programmatore, ma ne eleva il ruolo verso l'**Ingegneria Agantica** (*Agentic Engineering*). Il programmatore moderno coordina e supervisiona reti di agenti IA autonomi.

La Metafora: Il direttore d'orchestra non suona ogni singolo strumento dal vivo durante un concerto, ma possiede una conoscenza impeccabile della musica, del tempo, dell'armonia e degli spartiti. Senza la visione rigorosa del direttore, l'orchestra di robot e strumenti automatici suona in modo disarmonico e cade nel caos.

1.2 L'Evoluzione del Calcolo e l'Architettura di Von Neumann

1.2.1 La Pascalina e l'Informatica Meccanica (1642)

Agli albori del calcolo, l'algoritmo era letteralmente fuso nel metallo. Nella **Pascalina** inventata da Blaise Pascal, le ruote dentate e gli ingranaggi rappresentavano le operazioni aritmetiche. Per cambiare il calcolo o l'algoritmo, occorreva smontare e riprogettare fisicamente i ruotismi meccanici.

1.2.2 ENIAC e la Fisicità del Ricablaggio (Anni '40)

Con l'avvento dell'elettronica (le valvole termoioniche), l'**ENIAC** consentì calcoli infinitamente più veloci. Tuttavia, la riprogrammazione era ancora un'attività fisica estenuante: per cambiare



Figura 1.5: La Pascalina

programma, le ingegnere pioniere dovevano staccare e riattaccare fisicamente centinaia di cavi in enormi pannelli telefonici. *Riprogrammare significava letteralmente ricablare!*

1.2.3 John von Neumann e lo *Stored Program Concept* (1945)

La gigantesca intuizione del matematico John von Neumann fu il concetto di **Programma Memorizzato** (*Stored Program Concept*): memorizzare le istruzioni del programma nello stesso spazio fisico di memoria (la RAM) che contiene i dati su cui il programma opera.

1.2.4 La Macchina Universale (CPU vs RAM)

Grazie a questa architettura, l'hardware del computer è diventato immutabile e generico. La **CPU** (Central Processing Unit) esegue istruzioni in sequenza, mentre la **RAM** (Random Access Memory) ospita sia i dati che i programmi. Per trasformare il computer da calcolatrice a videogioco o elaboratore di testo, non serve più modificare i cavi: basta caricare sequenze di numeri differenti in memoria.

1.3 I Livelli di Astrazione: Dal Silicio al C



Figura 1.6: L'ENIAC ed il Ricablaggio

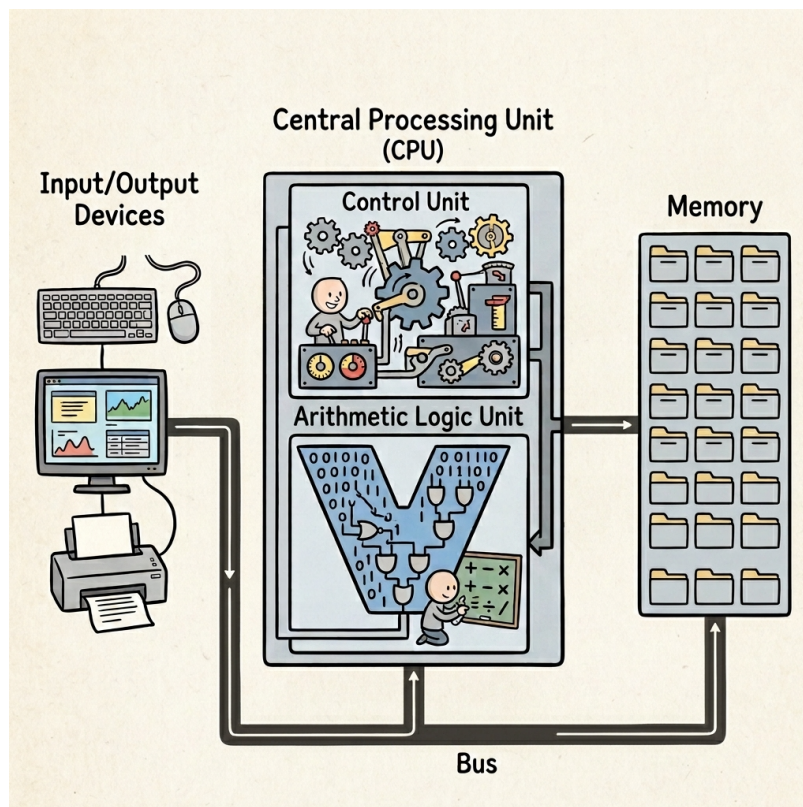


Figura 1.7: Von Neumann Architecture



Figura 1.8: I Livelli di Astrazione

1.3.1 Dal Codice Macchina al Linguaggio C

La macchina comprende ed esegue esclusivamente istruzioni binarie in **Codice Macchina** (sequenze di 0 ed 1 che corrispondono a livelli di tensione elettrica). Per non dover scrivere direttamente in binario, l'informatica ha creato diversi **livelli di astrazione**:

- **Livello Basso (Codice Macchina & Assembly):** Istruzioni binarie o mnemoniche rigide legate all'architettura specifica del chip (es. mov, add).
- **Livello Alto (Linguaggio C):** Esprime gli algoritmi in forma algebrica e strutturata, vicina al pensiero umano, astenendosi dai dettagli minuti del singolo processore.

1.3.2 La Metafora dell'Automobile e del Meccanico

Per comprendere l'importanza dei livelli di astrazione, pensiamo alla guida di un'automobile.

Per guidare un'auto ed andare da un punto A ad un punto B non ci serve sapere come funziona la camera di combustione o la fasatura dei pistoni: ci basta usare l'astrazione del volante e dei pedali (guidatore). Tuttavia, se la macchina si guasta o dobbiamo progettare un motore ad alte prestazioni, l'astrazione del volante non basta più: dobbiamo aprire il cofano e capire esattamente come interagiscono ingranaggi e pistoni (meccanico). Per risolvere un determinato problema in informatica serve sempre il giusto livello di astrazione!

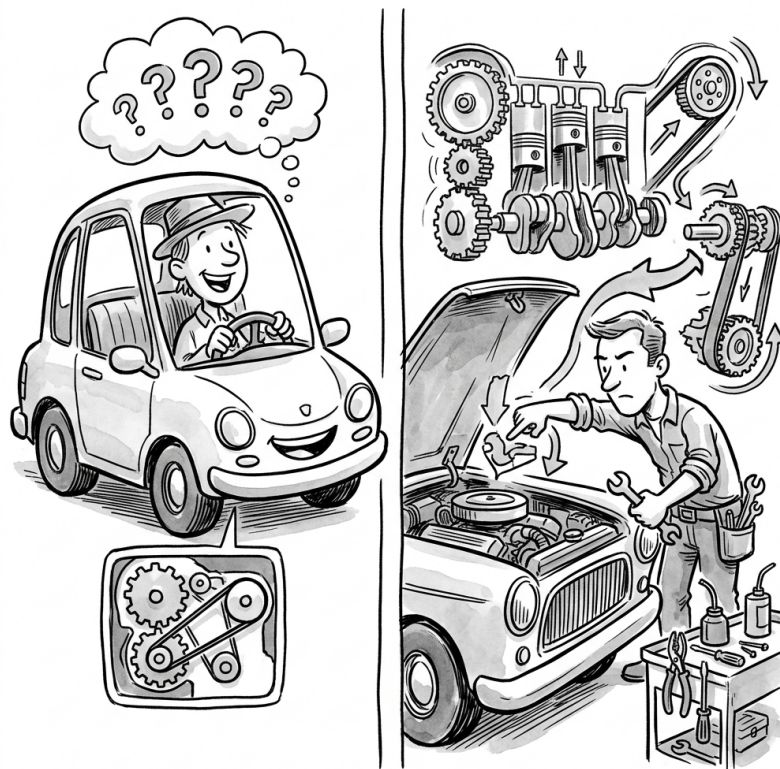


Figura 1.9: Metafora Guidatore e Meccanico

1.4 Sotto il Cofano: Registri e Traduzione Assembly

1.4.1 Il Collo di Bottiglia di Von Neumann ed i Registri CPU

Sebbene la CPU sia in grado di eseguire miliardi di operazioni al secondo, la RAM è un componente esterno collegato da linee fisiche (il **Bus**). Inviare e pescare continuamente dati dalla RAM ad ogni singola operazione crea il cosiddetto **Collo di Bottiglia di Von Neumann** (*Von Neumann Bottleneck*), che rallenterebbe l'elaborazione di oltre il 90%.

Per superare questo limite, i progettisti hanno saldato direttamente *all'interno* della CPU una manciata di celle di memoria velocissime chiamate **Registri**.

La CPU funziona come un operaio con le tasche: copia i dati necessari dalla RAM nelle proprie tasche (i registri), esegue il calcolo aritmetico istantaneo tramite la ALU (Arithmetic Logic Unit) e infine deposita il risultato in RAM.

1.4.2 Dal Codice Umano all'Assembly: L'illusione delle Variabili

Quando scriviamo in C un frammento come:

```
1 int main(void) {  
2     int x = 3;  
3     int y = 5;  
4     return x + y;  
5 }
```

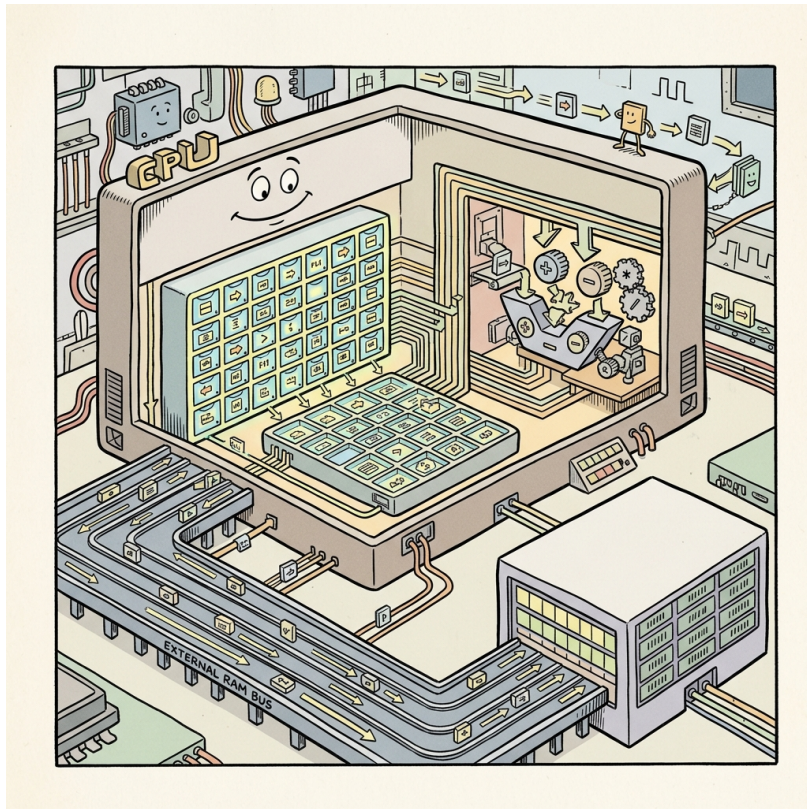


Figura 1.10: Processore basato su Registri Generali

Per la CPU le lettere x e y **non esistono**. Esse sono un'illusione ad alto livello creata dal linguaggio C per permetterci di ragionare in modo algebrico.

Guardiamo cosa genera il compilatore traducendo questo frammento in codice Assembly (sintassi Intel x86-64):

```
1 push    rbp                ; Salva il frame pointer precedente
   ↪ sullo stack
2 mov     rbp, rsp            ; Imposta il nuovo frame pointer per la
   ↪ funzione attuale
3 mov     DWORD PTR [rbp-4], 3 ; Copia il valore 3 nello spazio RAM di
   ↪ 'x'
4 mov     DWORD PTR [rbp-8], 5 ; Copia il valore 5 nello spazio RAM di
   ↪ 'y'
5 mov     edx, DWORD PTR [rbp-4] ; Carica 'x' dalla RAM al registro CPU
   ↪ edx
6 mov     eax, DWORD PTR [rbp-8] ; Carica 'y' dalla RAM al registro CPU
   ↪ eax
7 add     eax, edx            ; Somma il valore di edx nel registro
   ↪ eax
8 pop     rbp                ; Ripristina il frame pointer
   ↪ precedente
9 ret                                ; Restituisce il valore in eax al
   ↪ Sistema Operativo
```

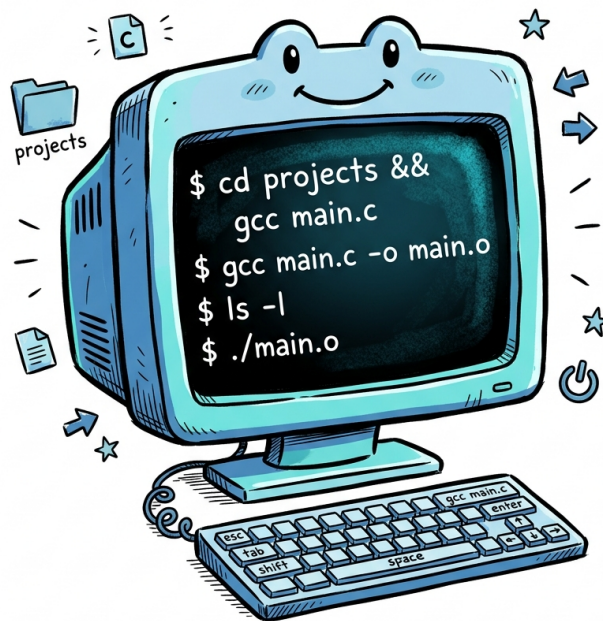



Figura 1.11: La riga di comando come intermediaria con il Sistema Operativo

Suggerimento

Strumento Utile: Compiler Explorer Potete sperimentare ed ispezionare in tempo reale come qualsiasi compilatore (gcc, clang) traduce il vostro codice C in Assembly utilizzando il celebre sito web interattivo [Compiler Explorer \(godbolt.org\)](https://godbolt.org).

1.5 La Riga di Comando: Parlare Direttamente al Sistema Operativo

Prima di addentrarci nella compilazione del codice, è fondamentale familiarizzare con lo strumento che ci permetterà di interagire direttamente con il computer: la **riga di comando** (o terminale).

1.5.1 Cos'è e a cosa serve

La riga di comando è un'interfaccia testuale che funge da intermediaria tra l'utente ed il Sistema Operativo (OS). A differenza delle classiche interfacce grafiche (GUI) a cui siamo abituati (fatte di finestre, icone e mouse), la riga di comando permette di impartire ordini digitando stringhe di testo ed eseguendo programmi o script in modo diretto ed efficiente.

Nell'era dello sviluppo moderno e dell'Intelligenza Artificiale, la riga di comando sta vivendo una grandissima rinascita di popolarità: gli assistenti di programmazione IA e gli agenti di coding autonomi operano ed eseguono strumenti quasi esclusivamente tramite interfacce

a riga di comando (CLI). Imparare ad utilizzarla è quindi una competenza chiave per ogni sviluppatore oggi.

1.5.2 Le Nostre Piattaforme: Shell Unix

In questo corso faremo riferimento allo standard delle shell di tipo **Unix** (come `bash` o `zsh`), che sono quelle preinstallate e utilizzate nativamente su **macOS** e **Linux**.

- **Windows:** Gli utenti Windows useranno il terminale fornito da **WSL** (Windows Subsystem for Linux), che mette a disposizione una shell Ubuntu Unix a tutti gli effetti.
- **Nota:** Windows possiede anche delle shell proprietarie (il Prompt dei comandi `cmd` e `PowerShell`), ma queste utilizzano comandi e sintassi differenti che non tratteremo in questo corso.

1.5.3 Comandi e Scorciatoie di Base

Per orientarsi ed operare agevolmente all'interno della shell Unix, è sufficiente memorizzare pochi semplici comandi e scorciatoie fondamentali riportati nella tabella seguente:

Tabella

Comando / Tasto	Nome completo	Descrizione
<code>pwd</code>	<i>Print Working Directory</i>	Stampa il percorso assoluto della cartella in cui ci si trova in questo momento.
<code>ls</code>	<i>List</i>	Elenca i file e le sottocartelle contenute nella cartella corrente.
<code>cd</code> <code><nome\{}_cartella></code>	<i>Change Directory</i>	Entra all'interno della cartella specificata.
<code>cd ..</code>	<i>Change Directory (Parent)</i>	Torna indietro di un livello, spostandosi nella cartella genitore.
<code>./<eseguibile></code>	<i>Execute local</i>	Esegue un file binario o un programma presente all'interno della cartella corrente (il prefisso <code>./</code> dice alla shell di cercare il file proprio lì).

Tab (tasto Tabulazione)	<i>Autocomplete</i>	Premuto mentre si scrive il nome di un file o di una cartella, ne autocompleta il nome risparmiando tempo ed evitando errori di battitura.
Freccia Su / Freccia Giù	<i>History</i>	Permette di scorrere la cronologia dei comandi precedentemente digitati e premere Invio per rieseguirli.
clear o Ctrl + L	<i>Clear screen</i>	Pulisce completamente l'interfaccia visiva del terminale, lasciando la riga di comando in alto.

1.6 Pratica al Terminale: La Magia del Valore di Ritorno (echo \$?)

1.6.1 Prima Compilazione e il Valore di Ritorno

Proviamo a scrivere ed eseguire questo codice. Creiamo un file chiamato `somma.c` con il nostro `main` che restituisce `x + y`. Compiliamo ed eseguiamo dal terminale:

```
1 gcc somma.c -o somma
2 ./somma
```

Dopo aver premuto Invio, notiamo che **a schermo non appare nulla**. Perché? Perché non abbiamo incluso alcuna istruzione esplicita di stampa a video (come `printf`).

1.6.2 Interrogare il Sistema Operativo con echo \$?

Dove è finito il risultato della nostra somma? L'istruzione `return x + y;` (ovvero l'istruzione `ret` in Assembly) ha consegnato il valore 8 direttamente nelle mani del **Sistema Operativo**! Possiamo interrogare il Sistema Operativo digitando immediatamente questo comando speciale nel terminale:

```
1 echo $?
```

Il terminale risponderà stampando a video il numero 8!

Suggerimento

Exit Status Il comando `echo $?` stampa l'ultimo **Exit Status** (codice di uscita) restituito dal processo appena terminato. Questo dimostra in modo tangibile che l'istruzione `return` nel `main` comunica l'esito dell'esecuzione direttamente al Sistema Operativo.



Figura 1.12: Linguaggi Interpretati

1.7 Dai Traduttori all'Hello World

1.7.1 Compilatori vs Interpreti

I linguaggi di programmazione si dividono principalmente in due grandi famiglie di traduzione:

Linguaggi Interpretati (Traduzione Simultanea)

Linguaggi come Python o JavaScript sono **interpretati**. Un programma chiamato *interprete* legge il codice sorgente riga per riga e lo traduce in comandi macchina *mentre* il programma è in esecuzione.

Linguaggi Compilati (La Fabbrica del C)

Il C è un linguaggio **compilato**. Un programma specializzato chiamato **compilatore** (`gcc` o `clang`) analizza e traduce l'intero codice sorgente in anticipo, producendo un file binario eseguibile nativo e velocissimo.

1.7.2 Anatomia della Toolchain del Compilatore (Le 4 Fasi)

La compilazione C non è un atto unico, ma un processo articolato in 4 fasi distinte:

Ipotizziamo di avere un file sorgente minimale chiamato `main.c` (che non include nemmeno `stdio.h` per evitare di importare migliaia di righe esterne) contenente questo codice:

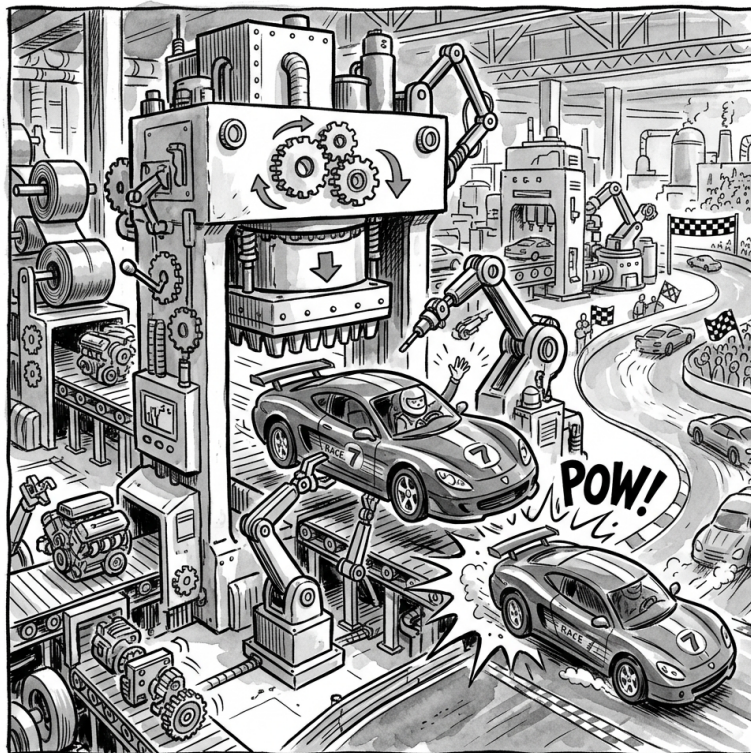


Figura 1.13: Linguaggi Compilati

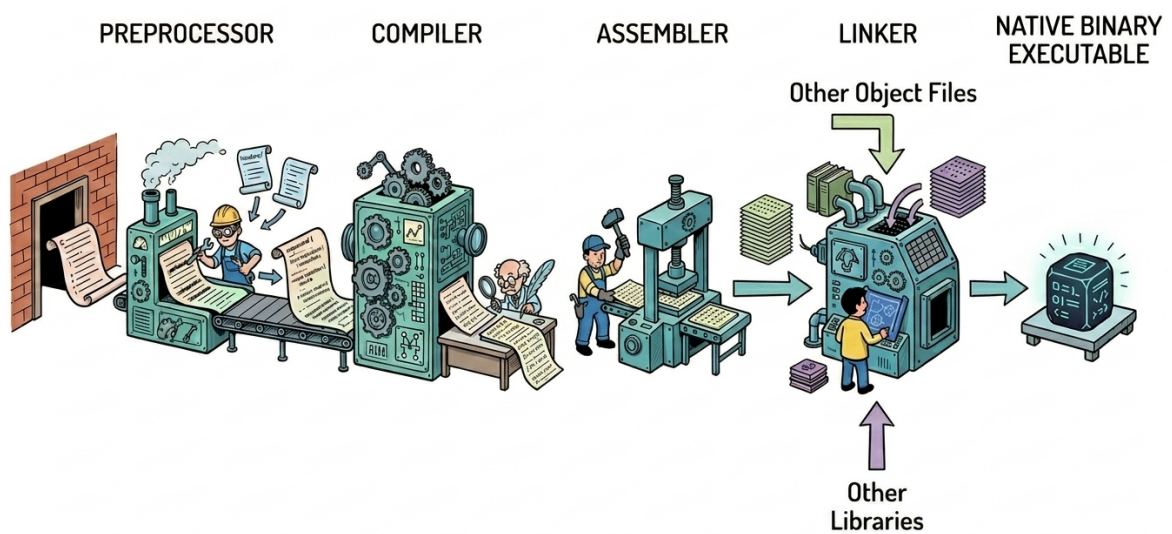


Figura 1.14: Toolchain di Compilazione C


```
1 int main(void) {  
2     int x = 3;  
3     int y = 5;  
4     return x + y;  
5 }
```

Vediamo i dettagli di ciascuna delle 4 fasi della toolchain, con i comandi per fermare il processo e ispezionare l'output generato:

1. **Preprocessore (cpp):** Analizza il testo, espande le direttive (es. `\{\}\#include`, `\{\}\#define`) ed elimina i commenti.

- **Comando:**

```
1 gcc -E main.c -o main.i
```

- **Risultato in main.i:**

```
1 # 1 "main.c"  
2 # 1 "<built-in>" 1  
3 # 1 "<built-in>" 3  
4 # 482 "<built-in>" 3  
5 # 1 "<command line>" 1  
6 # 1 "<built-in>" 2  
7 # 1 "main.c" 2  
8 int main(void) {  
9     int x = 3;  
10    int y = 5;  
11    return x + y;  
12 }
```

- **Spiegazione:** Le righe che iniziano con il simbolo `\{\}\#` (dette *linemarkers*) sono generate dal preprocessore per comunicare al compilatore le informazioni di tracciamento sull'origine delle righe e sui file inclusi. Poiché non abbiamo importato alcun file header, il codice sorgente rimane intatto.

2. **Compilatore (gcc/clang):** Traduce il codice C pre-elaborato in codice Assembly specifico per l'architettura hardware (in questo esempio, ARM64/Apple Silicon).

- **Comando:**

```
1 gcc -S main.i -o main.s
```

- **Risultato in main.s:**

```
1 .build_version macos, 26, 0 sdk_version 26, 4  
2 .section __TEXT,__text,regular,pure_instructions  
3 .globl _main ; -- Begin function  
4 ↪ main  
5 .p2align 2  
6 _main: ; @main  
7 .cfi_startproc
```

```

7 ; %bb.0:
8     sub sp, sp, #16
9     .cfi_def_cfa_offset 16
10    str wzr, [sp, #12]
11    mov w8, #3                                ; =0x3
12    str w8, [sp, #8]
13    mov w8, #5                                ; =0x5
14    str w8, [sp, #4]
15    ldr w8, [sp, #8]
16    ldr w9, [sp, #4]
17    add w0, w8, w9
18    add sp, sp, #16
19    ret
20    .cfi_endproc
21                                ; -- End function
22 .subsections_via_symbols

```

- **Spiegazione:** Questo codice è Assembly ARM64. Le righe che iniziano con il punto (es. `.section`, `.globl`) sono direttive per l'assembler. La funzione principale `{_main` alloca spazio sullo stack (`sub sp, sp, {#16}`), memorizza i valori 3 e 5 in memoria, li ricarica nei registri (`w8` e `w9`), esegue la somma (`add w0, w8, w9`) posizionando il risultato nel registro di ritorno `w0` (l'equivalente a 32 bit del registro `x0` per ARM), ripristina lo stack e ritorna con `ret`. L'elevato numero di metadati e direttive è tipico dei compilatori moderni per macchine Mac ARM.

3. **Assembler (as):** Traduce l'Assembly testuale in un file oggetto binario rilocabile (`main.o`) non ancora direttamente eseguibile.

- **Comando:**

```
1 gcc -c main.s -o main.o
```

- **Disassemblare per verificare:** Trattandosi di un file binario, non possiamo leggerlo direttamente come testo. Possiamo però usare uno strumento di disassemblaggio come `objdump` per convertire i byte binari in istruzioni assembly leggibili:

```
1 objdump -d main.o
```

- **Output di objdump:**

```

1 main.o: file format mach-o arm64
2
3 Disassembly of section __TEXT,__text:
4
5 0000000000000000 <tmp0>:
6      0: d10043ff      sub     sp, sp, #0x10
7      4: b9000fff      str     wzr, [sp, #0xc]
8      8: 52800068      mov     w8, #0x3                ; =3
9      c: b9000be8      str     w8, [sp, #0x8]
10     10: 528000a8      mov     w8, #0x5                ; =5
11     14: b90007e8      str     w8, [sp, #0x4]
12     18: b9400be8      ldr     w8, [sp, #0x8]
13     1c: b94007e9      ldr     w9, [sp, #0x4]

```

```

14      20: 0b090100      add     w0, w8, w9
15      24: 910043ff      add     sp, sp, #0x10
16      28: d65f03c0      ret

```

- **Spiegazione:** A sinistra vedete gli indirizzi e i codici operativi esadecimali (es. 52800068 per `mov w8, \{\}#3`) che la CPU decodifica ed esegue, e a destra la corrispondente istruzione assembly mnemonica.

4. **Linker (ld):** Unisce i file oggetto e le librerie di sistema per creare l'eseguibile binario finale (`main`), completo di intestazioni di caricamento per il Sistema Operativo.

- **Comando:**

```
1 gcc main.o -o main
```

- **Ispezionare il binario eseguibile:** Possiamo visualizzare l'intero file in formato esadecimale usando `hexdump`:

```
1 hexdump -C main
```

- **Output parziale di hexdump:**

```

1 00000000  cf fa ed fe 0c 00 00 01  00 00 00 00 02 00 00 00
   ↳ |.....|
2 00000010  10 00 00 00 e8 02 00 00  85 00 20 00 00 00 00 00
   ↳ |.....|
3 00000020  19 00 00 00 48 00 00 00  5f 5f 50 41 47 45 5a 45
   ↳ |...H...__PAGEZE|
4 00000030  52 4f 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |RO
   ↳ |.....|
5 00000040  00 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00
   ↳ |.....|
6 00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
   ↳ |.....|
7 00000060  00 00 00 00 00 00 00 00  19 00 00 00 e8 00 00 00
   ↳ |.....|
8 00000070  5f 5f 54 45 58 54 00 00  00 00 00 00 00 00 00 00 |
   ↳ |__TEXT.....|
9 00000080  00 00 00 00 01 00 00 00  00 40 00 00 00 00 00 00
   ↳ |.....@.....|
10 00000090  00 00 00 00 00 00 00 00  00 40 00 00 00 00 00 00
   ↳ |.....@.....|
11 000000a0  05 00 00 00 05 00 00 00  02 00 00 00 00 00 00 00
   ↳ |.....|
12 000000b0  5f 5f 74 65 78 74 00 00  00 00 00 00 00 00 00 00 |
   ↳ |__text.....|
13 000000c0  5f 5f 54 45 58 54 00 00  00 00 00 00 00 00 00 00 |
   ↳ |__TEXT.....|
14 000000d0  28 03 00 00 01 00 00 00  2c 00 00 00 00 00 00 00
   ↳ |(. . . . , . . . .|
15 000000e0  28 03 00 00 02 00 00 00  00 00 00 00 00 00 00 00
   ↳ |(. . . . .|
16 000000f0  00 04 00 80 00 00 00 00  00 00 00 00 00 00 00 00
   ↳ |.....|

```

```

17 00000100 5f 5f 75 6e 77 69 6e 64 5f 69 6e 66 6f 00 00 00 |
    ↳ __unwind_info...|
18 00000110 5f 5f 54 45 58 54 00 00 00 00 00 00 00 00 00 00 |
    ↳ __TEXT.....|
19 00000120 54 03 00 00 01 00 00 00 58 00 00 00 00 00 00 00 |T
    ↳ .....X.....|
20 00000130 54 03 00 00 02 00 00 00 00 00 00 00 00 00 00 00 |T
    ↳ .....|
21 00000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↳ |.....|
22 00000150 19 00 00 00 48 00 00 00 5f 5f 4c 49 4e 4b 45 44
    ↳ |...H...__LINKED|
23 00000160 49 54 00 00 00 00 00 00 00 40 00 00 01 00 00 00 |IT
    ↳ .....@.....|
24 00000170 00 40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 |.e
    ↳ .....@.....|
25 00000180 c8 01 00 00 00 00 00 00 01 00 00 00 01 00 00 00
    ↳ |.....|
26 00000190 00 00 00 00 00 00 00 00 34 00 00 80 10 00 00 00
    ↳ |.....4.....|
27 000001a0 00 40 00 00 38 00 00 00 33 00 00 80 10 00 00 00 |.e
    ↳ ..8...3.....|
28 000001b0 38 40 00 00 30 00 00 00 02 00 00 00 18 00 00 00 |8e
    ↳ ..0.....|
29 000001c0 70 40 00 00 02 00 00 00 90 40 00 00 20 00 00 00 |pe
    ↳ .....@.. ...|
30 000001d0 0b 00 00 00 50 00 00 00 00 00 00 00 00 00 00 00
    ↳ |...P.....|
31 000001e0 00 00 00 00 02 00 00 00 02 00 00 00 00 00 00 00
    ↳ |.....|
32 000001f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↳ |.....|
33
34 *
35 00000220 0e 00 00 00 20 00 00 00 0c 00 00 00 2f 75 73 72
    ↳ |.... ...../usr|
36 00000230 2f 6c 69 62 2f 64 79 6c 64 00 00 00 00 00 00 00 |/
    ↳ lib/dyld.....|
37 00000240 1b 00 00 00 18 00 00 00 c6 3c 4a 2f 9b dd 3f 27
    ↳ |.....<J/..?'|
38 00000250 af 05 1a cb 5a 82 0f e5 32 00 00 00 20 00 00 00
    ↳ |...Z...2... ...|
39 00000260 01 00 00 00 00 00 1a 00 00 04 1a 00 01 00 00 00
    ↳ |.....|
40 00000270 03 00 00 00 00 08 f2 04 2a 00 00 00 10 00 00 00
    ↳ |.....*.....|
41 00000280 00 00 00 00 00 00 00 00 28 00 00 80 18 00 00 00
    ↳ |.....(.....|
42 00000290 28 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↳ |(.....|
43 000002a0 0c 00 00 00 38 00 00 00 18 00 00 00 02 00 00 00
    ↳ |...8.....|
44 000002b0 00 00 4c 05 00 00 01 00 2f 75 73 72 2f 6c 69 62 |..L
    ↳ ...../usr/lib|
45 000002c0 2f 6c 69 62 53 79 73 74 65 6d 2e 42 2e 64 79 6c |/
    ↳ libSystem.B.dyl|

```



```

46 000002d0 69 62 00 00 00 00 00 00 26 00 00 00 10 00 00 00 |ib
    ↳ .....&.....|
47 000002e0 68 40 00 00 08 00 00 00 29 00 00 00 10 00 00 00 |he
    ↳ .....).....|
48 000002f0 70 40 00 00 00 00 00 00 1d 00 00 00 10 00 00 00 |pe
    ↳ .....|
49 00000300 b0 40 00 00 18 01 00 00 00 00 00 00 00 00 00 00 |.e
    ↳ .....|
50 00000310 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↳ |.....|
51 00000320 00 00 00 00 00 00 00 00 ff 43 00 d1 ff 0f 00 b9
    ↳ |.....C.....|
52 00000330 68 00 80 52 e8 0b 00 b9 a8 00 80 52 e8 07 00 b9 |h..
    ↳ R.....R....|
53 00000340 e8 0b 40 b9 e9 07 40 b9 00 01 09 0b ff 43 00 91 |..@
    ↳ ...@.....C..|
54 00000350 c0 03 5f d6 01 00 00 00 1c 00 00 00 00 00 00 00 |.._
    ↳ .....|
55 00000360 1c 00 00 00 00 00 00 00 1c 00 00 00 02 00 00 00
    ↳ |.....|
56 00000370 28 03 00 00 40 00 00 00 40 00 00 00 54 03 00 00
    ↳ |(..@...@...T...|
57 00000380 00 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00
    ↳ |...@.....|
58 00000390 00 00 00 00 03 00 00 00 0c 00 01 00 10 00 01 00
    ↳ |.....|
59 000003a0 00 00 00 00 00 10 00 02 00 00 00 00 00 00 00 00
    ↳ |.....|
60 000003b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↳ |.....|
61
62 *
63 00004000 00 00 00 00 20 00 00 00 30 00 00 00 30 00 00 00
    ↳ |.... ..0...0...|
64 00004010 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
    ↳ |.....|
65 00004020 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↳ |.....|
66 00004030 00 00 00 00 00 00 00 00 00 01 5f 00 12 00 00 00
    ↳ |....._.....|
67 00004040 00 02 00 00 00 03 00 a8 06 00 00 02 5f 6d 68 5f
    ↳ |....._mh_|
68 00004050 65 78 65 63 75 74 65 5f 68 65 61 64 65 72 00 09 |
    ↳ execute_header..|
69 00004060 6d 61 69 6e 00 0d 00 00 a8 06 00 00 00 00 00 00 |
    ↳ main.....|
70 00004070 02 00 00 00 0f 01 10 00 00 00 00 00 01 00 00 00
    ↳ |.....|
71 00004080 16 00 00 00 0f 01 00 00 28 03 00 00 01 00 00 00
    ↳ |.....(.....|
72 00004090 20 00 5f 5f 6d 68 5f 65 78 65 63 75 74 65 5f 68 |.
    ↳ __mh_execute_h|
73 000040a0 65 61 64 65 72 00 5f 6d 61 69 6e 00 00 00 00 00 |
    ↳ eader._main.....|
74 000040b0 fa de 0c c0 00 00 01 11 00 00 00 01 00 00 00 00
    ↳ |.....|

```

```

75 000040c0 00 00 00 14 fa de 0c 02 00 00 00 fd 00 02 04 00
    ↪ |.....|
76 000040d0 00 02 00 02 00 00 00 5d 00 00 00 58 00 00 00 00
    ↪ |.....]...X...|
77 000040e0 00 00 00 05 00 00 40 b0 20 02 00 0c 00 00 00 00
    ↪ |.....@. ....|
78 000040f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↪ |.....|
79
80 *
81 00004110 00 00 00 2c 00 00 00 00 00 00 00 01 6d 61 69 6e
    ↪ |...,.....main|
82 00004120 00 1b 34 0e f5 0e 6a a2 a6 9d c2 8a df f8 2b 80
    ↪ |..4...j.....+.|
83 00004130 3e 5c 27 2b 4d 63 c6 36 90 4c d6 80 e4 12 08 fc
    ↪ |>\'+Mc.6.L.....|
84 00004140 af ad 7f ac b2 58 6f c6 e9 66 c0 04 d7 d1 d1 6b
    ↪ |.....Xo..f.....k|
85 00004150 02 4f 58 05 ff 7c b4 7c 7a 85 da bd 8b 48 89 2c |.0X
    ↪ ..|. |z....H.,|
86 00004160 a7 ad 7f ac b2 58 6f c6 e9 66 c0 04 d7 d1 d1 6b
    ↪ |.....Xo..f.....k|
87 00004170 02 4f 58 05 ff 7c b4 7c 7a 85 da bd 8b 48 89 2c |.0X
    ↪ ..|. |z....H.,|
88 00004180 a7 ad 7f ac b2 58 6f c6 e9 66 c0 04 d7 d1 d1 6b
    ↪ |.....Xo..f.....k|
89 00004190 02 4f 58 05 ff 7c b4 7c 7a 85 da bd 8b 48 89 2c |.0X
    ↪ ..|. |z....H.,|
90 000041a0 a7 66 32 27 13 70 79 29 4b 7b 1c 83 53 97 b1 da |.f2
    ↪ '.py)K{..S...|
91 000041b0 09 de 2e 98 15 a8 bb 2c 87 ec eb 22 55 ca e6 0c
    ↪ |.....,"U...|
92 000041c0 40 00 00 00 00 00 00 00
    ↪ .....|
93 000041c8

```

- **Spiegazione:** Questo dump esadecimale (generato in formato Mach-O a 64-bit per Apple Silicon Mac) rappresenta l'eseguibile completo. All'offset 00000320 potete notare i reali byte macchina binari che abbiamo disassemblato nella fase precedente (es. ff 43 00 d1, ff 0f 00 b9, ecc.), ora inseriti in una struttura file strutturata insieme a metadati di caricamento, percorsi per il caricatore dinamico delle librerie (/usr/lib/dyld e /usr/lib/libSystem.B.dylib) e tabelle dei simboli generati dal linker.

1.7.3 Il Problema del Mondo Esterno (stdio.h)

La CPU da sola conosce solo l'aritmetica ed i registri; non possiede alcuna nozione su come accedere al monitor, formattare caratteri grafici o leggere dalla tastiera. Per fare ciò, il programma deve interagire con il Sistema Operativo richiedendo l'ausilio delle librerie di sistema.

Il linguaggio C è improntato alla massima efficienza: non include di default alcuna funzione che non sia stata espressamente richiesta. Per abilitare le funzioni di Input/Output occorre includere l'intestazione standard tramite la direttiva `\{\}#include <stdio.h>`.

1.8 Anatomia dell'Hello World e i Commenti nel Codice

1.8.1 Anatomia del Primo Programma C Reale

Esaminiamo ora il classico primo programma C, arricchito dai commenti esplicativi:

```
1 // Questo ? un commento a riga singola (introdotto nello standard C99)
2 #include <stdio.h> /* Direttiva al preprocessore per l'I/O standard */
3
4 /*
5     Questo ? un commento multi-riga.
6     I commenti vengono totalmente ignorati dal compilatore
7     ed eliminati nella prima fase dal Preprocessore!
8 */
9 int main(void) {
10     printf("Benvenuti a Programmazione 1!\n"); // Stampa la stringa su
        ↪ schermo
11     return 0; // Restituisce il valore 0 al Sistema Operativo (Successo)
12 }
```

Suggerimento

Curiosità: Il Preprocessore in Hello World Per completezza, provate a lanciare il preprocessore su questo file che include `<stdio.h>` eseguendo:

```
1 cc hello.c -E -o hello.i
```

Se aprite il file `hello.i` risultante con un editor di testo, noterete che contiene migliaia di righe di codice (spesso più di 20.000 righe). Questo accade perché la direttiva `\{\}#include <stdio.h>` viene espansa dal preprocessore che copia letteralmente nel file tutte le definizioni e le dichiarazioni necessarie alla libreria di input/output standard prima dell'avvio del compilatore.

1.8.2 Sintassi dei Commenti in C

- **// Commento a riga singola:** tutto ciò che si trova dopo i due slash sulla medesima riga viene ignorato.
- **/* ... */ Commento multi-riga:** ignora tutto il testo compreso tra il simbolo di apertura `/*` ed il simbolo di chiusura `*/`, anche su più righe.

1.8.3 Il Punto e Virgola (;) come Terminatore

Nelle righe di codice `printf(...)` e `return 0;`, noterete la presenza costante del punto e virgola `;`.

- In C, il punto e virgola non è un separatore di istruzioni (come in altri linguaggi), ma un **terminatore di istruzione** (statement terminator).
- Indica al compilatore che un'istruzione è formalmente conclusa. Dimenticarlo è uno degli errori di sintassi più comuni e genera immediati errori in fase di compilazione.

1.8.4 Whitespace e Formattazione Libera (Free-form)

Il compilatore C ignora completamente tutti i caratteri invisibili come spazi, tabulazioni e ritorni a capo (detti **whitespace**), ad eccezione di quelli inseriti all'interno dei letterali stringa (il testo delimitato da virgolette "..."). Teoricamente, potreste scrivere l'intero programma `main` su un'unica riga:

```
1 int main(void) { printf("Benvenuti!"); return 0; }
```

Il compilatore produrrebbe lo stesso identico eseguibile binario nativo.

1.8.5 L'Importanza dell'Indentazione

Se il compilatore ignora gli spazi, perché ci preoccupiamo di organizzare il codice su più righe e con rientri precisi?

- **Per la leggibilità umana:** L'indentazione (l'uso di spazi o tabulazioni per rientrare ad ogni blocco racchiuso tra parentesi graffe `{ } { }`) è una convenzione fondamentale e una questione di stile condivisa.
- Consente a voi e ai vostri colleghi di leggere e interpretare all'istante la struttura logica del codice. Scrivere codice privo di indentazione rende la diagnosi degli errori estremamente complessa e pruriginosa.

1.8.6 Verifica dell'Hello World con `echo $?`

Per compilare il nostro codice utilizzeremo il comando `gcc`.

Suggerimento

Prendere buone abitudini: Il flag `-Wall` Quando compilate, abituatevi fin dal primo giorno ad aggiungere il flag `-Wall` (*Warnings: All*):

```
1 gcc -Wall hello.c -o hello
```

Questo comando dice al compilatore di non limitarsi a segnalare errori bloccanti, ma di avvisarvi con dei messaggi di avvertimento (*warnings*) su comportamenti strani o rischiosi nel vostro codice. Risparmierete ore di debugging!

Eseguiamo il programma appena generato:

```
1 ./hello
```

Output a schermo:

```
1 Benvenuti a Programmazione 1!
```

Se ora verifichiamo l'Exit Status restituito al Sistema Operativo:

1 **echo** \$?

Il terminale risponderà 0.

Nota

Convenzione di Successo In tutti i sistemi operativi POSIX (Linux, macOS, Unix), l'Exit Status 0 rappresenta la convenzione universale per indicare che il programma è stato completato con **successo e senza errori**. Qualsiasi valore diverso da zero (es. 1, 2, 139) indica un codice di errore specifico.

1.9 Quiz di Autovalutazione

Mettetevi alla prova con questo test rapido per verificare la vostra comprensione dei concetti di questa lezione. Cliccate su ciascuna domanda per verificare la risposta corretta e la spiegazione.

Quiz 1

Domanda

1. Nell'architettura di von Neumann, quale componente si occupa di coordinare l'intero sistema recuperando le istruzioni dalla memoria ed eseguendole?

- A) La memoria RAM.
- B) L'Unità di Controllo (Control Unit - CU) all'interno della CPU.
- C) L'Unità Aritmetico Logica (ALU).
- D) Il bus di sistema per il trasferimento dei dati.

Soluzione
Risposta corretta: B
Spiegazione: La Control Unit (CU) recupera le istruzioni dalla RAM tramite il ciclo di Fetch-Decode-Execute e coordina le altre componenti hardware del sistema.

Quiz 2

Domanda

2. Dove risiedono le variabili ed il codice di un programma mentre è in corso di esecuzione?

- A) Esclusivamente all'interno dei registri della CPU per tutta la durata del programma.
- B) Nella memoria RAM.
- C) Sull'hard disk o SSD.

- D) All'interno del file header `stdio.h`.

Soluzione
Risposta corretta: B
Spiegazione: La RAM ospita sia il codice del programma che i dati dinamici (variabili) durante il tempo di esecuzione (*runtime*). I registri contengono solo i dati correnti calcolati in quel preciso istante.

Quiz 3

Domanda

3. Qual è la differenza principale tra un linguaggio compilato (come il C) ed uno interpretato (come Python)?

- A) I linguaggi compilati sono tradotti in codice macchina nativo prima dell'esecuzione; quelli interpretati sono letti riga per riga a runtime da un programma interprete.
- B) I linguaggi compilati non possono gestire input dall'utente.
- C) I linguaggi interpretati sono più veloci nell'esecuzione su microcontrollori.
- D) I linguaggi compilati non necessitano di una funzione `main`.

Soluzione
Risposta corretta: A
Spiegazione: Il compilatore C traduce in blocco il codice in linguaggio macchina proprietario della CPU. Un interprete invece legge e converte il codice riga per riga durante l'esecuzione, introducendo un sovraccarico di prestazioni.

Quiz 4

Domanda

4. Che cos'è il linker all'interno della toolchain di compilazione?

- A) Il software che rileva gli errori di sintassi nel codice sorgente.
- B) Il programma che unisce il codice oggetto generato dal compilatore con le librerie esterne precompilate per creare un eseguibile finale.
- C) L'editor di testo utilizzato per scrivere il codice.
- D) La componente hardware che sposta i dati tra i registri ed il bus.

Soluzione
Risposta corretta: B
Spiegazione: Il linker unisce i file oggetto (.o) prodotti dal compilatore ed eventuali librerie esterne (es. stdio del C) risolvendo tutti i riferimenti agli indirizzi delle funzioni per generare il file eseguibile finale.

Quiz 5

Domanda

5. A cosa serve la direttiva `\{\}\#include <stdio.h>` posta in cima a un programma C?

- A) A ordinare al compilatore di allocare 8 byte di memoria nello Stack.
- B) A importare il prototipo delle funzioni della libreria standard di Input/Output (come `printf`).
- C) A dichiarare la funzione principale `main`.
- D) A azzerare il valore di ritorno del programma.

Soluzione
Risposta corretta: B
Spiegazione: `\{\}\#include <stdio.h>` è una direttiva del preprocessore che inserisce l'header file `stdio.h`, contenente le dichiarazioni delle funzioni base per interagire con lo schermo e la tastiera.

Quiz 6

Domanda

6. Cosa significa il parametro `void` tra le parentesi tonde in `int main(void)`?

- A) Che il programma restituisce un valore nullo.
- B) Che la funzione `main` non accetta alcun argomento o parametro in ingresso.
- C) Che il programma contiene un errore di sintassi.
- D) Che la funzione principale alloca solo puntatori a memoria Heap.

Soluzione
Risposta corretta: B
Spiegazione: In C, specificare `void` all'interno della lista dei parametri di una funzione indica in modo esplicito che la funzione non accetta parametri.

Quiz 7

Domanda

7. In un sistema Unix/macOS, cosa indica un valore di exit status uguale a 0 restituito da un programma?

- A) Che il programma è crashato prima dell'esecuzione.
- B) Che il programma è terminato con successo e senza errori.
- C) Che non è stato stampato alcun testo a schermo.
- D) Che la memoria RAM è stata riempita completamente.

Soluzione
Risposta corretta: B
Spiegazione: Per convenzione POSIX, l'exit status 0 rappresenta la conclusione ottimale del programma. Qualsiasi valore diverso da zero indica codici di errore definiti dallo sviluppatore.

Quiz 8

Domanda

8. Quale comando da terminale permette di ispezionare l'exit status restituito dall'ultimo programma eseguito?

- A) `cat result`
- B) `echo $?`
- C) `gcc -o output`
- D) `pwd`

Soluzione
Risposta corretta: B
Spiegazione: Nei terminali Bash o Zsh, la variabile speciale `$?` contiene il codice di uscita dell'ultimo processo terminato. Usando `echo $?` se ne visualizza il valore a schermo.

1.10 Esercizi Pratici

Esercizio 1: Il Primo Eseguitibile

Domanda

Create un file chiamato `saluto.c`. Scrivete un programma in C che stampi a schermo il testo *"Benvenuti al corso di Programmazione 1 - A.A. 2026/2027!"* seguito da una nuova riga. Compilate il codice da terminale generando un eseguibile chiamato `saluto` ed

eseguitelo.

```
1 gcc saluto.c -o saluto
2 ./saluto
```

Comandi da lanciare nel terminale:

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("Benvenuti al corso di Programmazione 1 - A.A.
    ↪ 2026/2027!\n");
5     return 0;
6 }
```

Codice sorgente saluto.c:

Soluzione

Esercizio 2: Ispezione dell'Exit Status di Errore

Domanda

Modificate l'istruzione `return 0;` del vostro programma dell'Esercizio 1 inserendo un valore di uscita a vostra scelta diverso da zero (ad esempio `return 42;`). Compilate nuovamente ed eseguite. Verificate tramite terminale che il sistema operativo riceva correttamente il nuovo codice di uscita personalizzato.

Il terminale risponderà stampando il valore 42.

```
1 gcc saluto.c -o saluto
2 ./saluto
3 echo $?
```

Comandi da terminale per verificare:

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("Benvenuti al corso di Programmazione 1 - A.A.
    ↪ 2026/2027!\n");
5     return 42; // Restituiamo un codice di uscita personalizzato
6 }
```

Codice sorgente modificato:

Soluzione

Esercizio 3: Stampa Multilinea

Domanda

Scrivete un programma in C che stampi a schermo un rettangolo di asterischi di dimensione 5 colonne per 3 righe, utilizzando una sola chiamata alla funzione `printf`. *Suggerimento: ricordatevi di inserire il carattere speciale di ritorno a capo `\n` nei punti strategici della stringa.*

```
1 gcc rettangolo.c -o rettangolo
2 ./rettangolo
```

Comandi da lanciare nel terminale:

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("*****\n*****\n*****\n");
5     return 0;
6 }
```

Codice sorgente rettangolo.c:

Soluzione

Esercizio 4: Trovare il Comando nel System Path

Domanda

Utilizzando il terminale (su macOS o Linux), determinate in quale cartella di sistema si trova l'eseguibile del compilatore `gcc`. *Suggerimento: utilizzate il comando `which` oppure `whereis`.*

Spiegazione: Il comando `which` analizza le directory elencate nella variabile d'ambiente `$PATH` del sistema operativo e restituisce il percorso assoluto del primo file eseguibile corrispondente al nome specificato (es. `/usr/bin/gcc`).

```
1 which gcc
```

Comando da lanciare nel terminale:

Soluzione

Esercizio 5: Scrittura di Commenti e Codice

Domanda

Auto-esplicativo

Scrivete un programma minimale in C che non esegua alcuna stampa ma che restituisca al sistema operativo il codice di uscita 12. All'interno del file, inserite sia un commento a riga singola (`//`) sia un commento multilinea (`/* ... */`) per documentare lo scopo del programma. Compilate e verificate l'exit status tramite terminale.

Soluzione

Codice sorgente commenti.c:

```

1 #include <stdio.h>
2
3 /*
4  * Corso di Programmazione 1
5  * Questo programma dimostra l'uso dei commenti multilinea
6  * e restituisce l'exit status 12.
7  */
8
9 int main(void) {
10     // Restituiamo 12 per simulare un exit status di errore
11     ↪ specifico
12     return 12;
13 }
```

Comandi da lanciare nel terminale:

```

1 gcc commenti.c -o commenti
2 ./commenti
3 echo $?
```

Il terminale risponderà stampando il valore 12.

Esercizio 6: Compilazione con Avvisi (Warnings)**Domanda**

Provate a compilare un file C in cui dichiarate una variabile senza usarla, forzando il compilatore a segnalare tutti gli avvisi possibili tramite l'opzione `-Wall`. Scrivete i comandi da lanciare nel terminale.

Spiegazione: Il flag `-Wall` attiva quasi tutti i messaggi di avvertimento (*warnings*) del compilatore. Nel compilatore GCC/Clang, compilando questo file apparirà un avviso simile a: `warning: unused variable 'x',`

```

1 gcc -Wall warning.c -o warning
```

Comando di compilazione:

```

1 #include <stdio.h>
2
3 int main(void) {
4     int x = 10; // Variabile dichiarata ma inutilizzata
5     return 0;
6 }
```

Soluzione

Codice sorgente warning.c:

Esercizio 7: Compilazione di file sorgenti multipli**Domanda**

Immaginate di avere due file sorgenti separati: `main.c` (contenente la funzione `main`) e `helper.c` (contenente funzioni di supporto). Scrivete la sequenza di comandi GCC per compilarli prima in file oggetto (.o) e poi collegarli (linking) in un unico file eseguibile chiamato `app`.

```

1 # Compila i singoli sorgenti in file oggetto senza fare il linking
  ↪ (-c)
2 gcc -c main.c -o main.o
3 gcc -c helper.c -o helper.o
4
5 # Collega i file oggetto in un unico eseguibile finalizzato
6 gcc main.o helper.o -o app

```

Comandi da lanciare nel terminale:

Soluzione

Esercizio 8: Errore nel preprocessore**Domanda**

Scrivete un codice C volutamente errato in cui omettete il simbolo `\{\}\#` davanti alla direttiva `include <stdio.h>`. Compilatelo con `gcc` e analizzate l'errore del compilatore. Spiegate perché fallisce in fase di preprocessing.

Spiegazione: Il preprocessore analizza solo le righe che iniziano con `\{\}\#`. Senza questo carattere, la riga viene vista dal compilatore vero e proprio come normale codice C. Il compilatore cerca quindi di interpretare `include` come il tipo di una variabile o funzione, fallendo immediatamente poiché non riconosce la parola chiave.

Errore generato da GCC: error: unknown type name 'include' o error: expected '=', ',', ';', '...',

```

1 include <stdio.h> // Errore: manca il simbolo '\{\}\#'
2
3 int main(void) {
4     printf("Test errore\n");
5     return 0;
6 }

```

Codice sorgente errato `err\{\}\_preprocessore.c`:

Soluzione

Esercizio 9: La Sequenza di Escape per il Backslash**Domanda**

Scrivete un programma in C che stampi a schermo il percorso assoluto tipico di Windows: C:\Program Files\MyCode\ . Spiegate come gestire i caratteri di backslash per evitare che vengano interpretati come sequenze di escape (es. \P, \M).

Spiegazione: In C, il carattere backslash \ è un carattere di escape che introduce comandi speciali (come \n per andare a capo o \t per tabulare). Per stampare a schermo un carattere \ letterale, è necessario effettuarne l'escape raddoppiandolo: \\.

```
1 gcc percorso.c -o percorso
2 ./percorso
```

Comandi da lanciare nel terminale:

```
1 #include <stdio.h>
2
3 int main(void) {
4     // Usiamo il doppio backslash '\\' per dire a printf di
5     printf("C:\\Program Files\\MyCode\\n");
6     return 0;
7 }
```

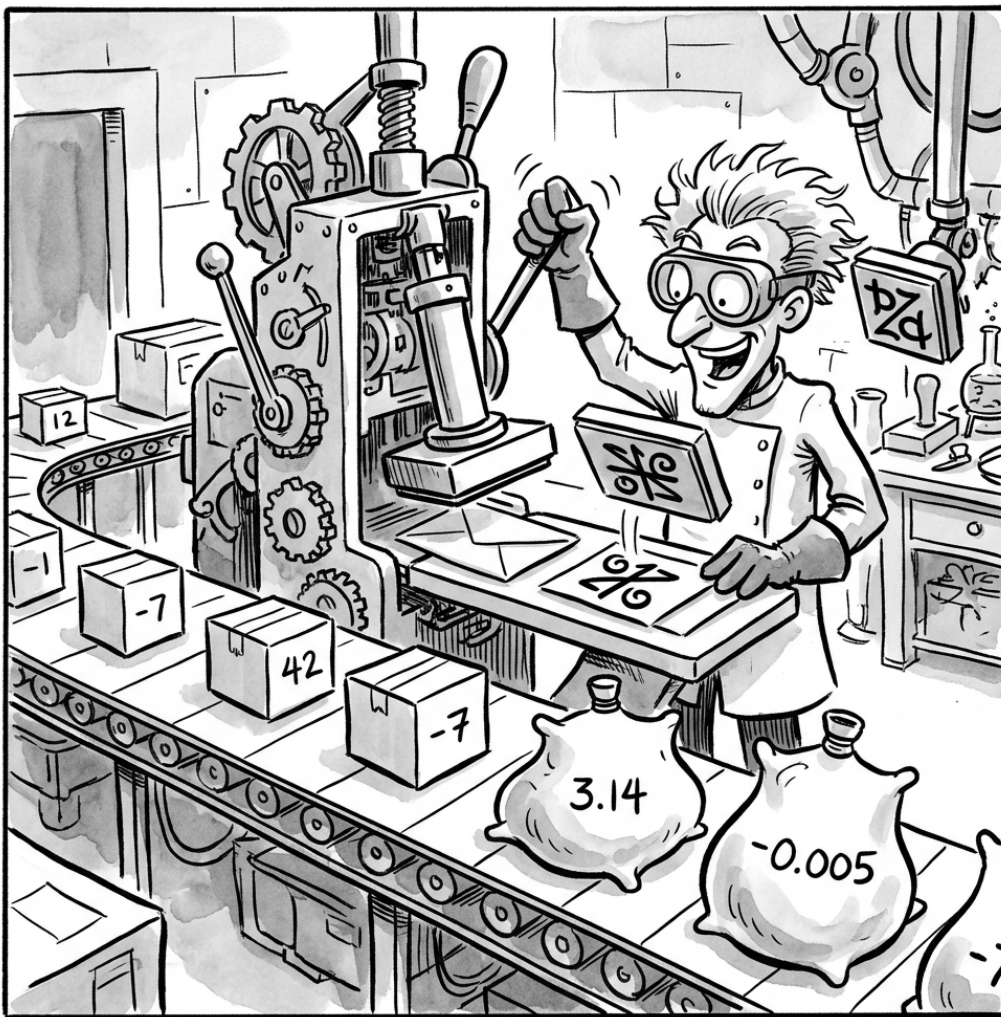
↪ stampare un singolo '\\', testuale.

Codice sorgente percorso.c:

Soluzione

LEZIONE 2

La Logica dei Numeri e l'Input/Output



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 2

La Logica dei Numeri e l'Input/Output

Nelle lezioni precedenti abbiamo visto come un programma minimale possa restituire un valore numerico al sistema operativo o visualizzare messaggi statici sullo schermo. In questa lezione faremo un importante passo avanti: introdurremo il concetto di variabile, scopriremo come interagire dinamicamente con l'utente chiedendo dati da tastiera tramite `scanf`, e come scrivere espressioni logiche ed algebriche per guidare le scelte fondamentali della CPU tramite il costrutto `if`.

2.1 Variabili, Tipi di Dato e I/O Standard

2.1.1 Che cos'è una Variable sotto il cofano?

Per un programmatore alle prime armi, una variabile assomiglia a una lettera algebrica (come la x o la y delle equazioni scolastiche). Sotto il cofano del computer, tuttavia, una variabile è una **denominazione simbolica associata a un'area fisica della memoria RAM**.

Quando scrivete nel vostro codice C:

```
1 int age = 19;
```

State ordinando al compilatore di:

1. Riservare una cella fisica di memoria (dello Stack) di dimensione fissa (di norma **4 byte** per il tipo `int` sui sistemi moderni a 64 bit).
2. Associare il nome simbolico `age` all'indirizzo di partenza di quella cella.
3. Scrivere dentro quella cella la sequenza binaria corrispondente al numero intero 19 (00000000 00000000 00000000 00010011).

Notiamo che i nomi delle variabili sono un'astrazione creata esclusivamente per agevolare lo sviluppatore umano. Nel momento in cui il compilatore traduce il codice in linguaggio Assembly e in codice macchina, tutti i nomi simbolici (`age`, `raggio`, etc.) scompaiono del tutto, venendo sostituiti da indirizzi di memoria fisici o scostamenti relativi a registri di base (es. `[rbp-4]`).

2.1.2 Il Concetto di Tipo

Il **tipo di dato** indica al compilatore due informazioni cruciali:

1. **La dimensione** in byte della scatola di memoria da riservare.
2. **La codifica** (la regola di traduzione) con cui interpretare i bit salvati all'interno. Ad esempio, la sequenza di bit che rappresenta l'intero 65 è identica a quella che rappresenta il carattere 'A' in codifica ASCII; solo il tipo ci dice come interpretarla.

Il linguaggio C impone la dichiarazione esplicita del tipo per ragioni di efficienza compilativa ed esecutiva: conoscendo in anticipo il tipo, il compilatore genera direttamente le istruzioni macchina native più corrette ed efficienti (la somma tra interi `add` usa circuiti della CPU radicalmente diversi rispetto alla somma in virgola mobile per numeri reali).

2.1.3 Interi vs Reali: `int` e `double`

All'inizio del nostro viaggio ci concentreremo su due soli tipi fondamentali:

- **`int`**: Rappresenta i numeri interi con segno (es. 0, 42, -1053). Di norma occupa **4 byte** (32 bit), consentendo di rappresentare numeri nell'intervallo tra circa -2 miliardi e +2 miliardi.
- **`double`**: Rappresenta i numeri reali o decimali (es. 3.14, -0.005, 2.0). Il nome deriva da *double precision* (doppia precisione): occupa **8 byte** (64 bit) e segue lo standard internazionale **IEEE 754** per la codifica in virgola mobile. È il tipo di default utilizzato dal C quando scrivete una costante decimale nel codice.

2.1.4 Dichiarazioni con Inizializzazione e Dichiarazioni Disgiunte

In C potete decidere di dichiarare e inizializzare una variabile contemporaneamente sulla stessa riga:

```
1 int age = 19; // Alloca la memoria e vi scrive subito 19
```

Oppure potete scegliere la strada della dichiarazione disgiunta dall'assegnamento:

```
1 int score; // Dichiarazione: lo spazio viene riservato, ma il valore ?
   ↪ indeterminato
2 score = 100; // Assegnamento successivo: viene scritto 100
```

Il Pericolo delle Variabili Non Inizializzate

Se leggete il valore di una variabile locale dichiarata ma non ancora inizializzata (es. stampare `score` subito dopo `int score;`), il programma non darà errore di compilazione ma conterrà un valore del tutto imprevedibile, comunemente chiamato **valore spazzatura** (*garbage value*). La cella di memoria conterrà semplicemente i residui binari di qualunque operazione fosse stata eseguita precedentemente in quell'area della RAM. Inizializzate sempre le vostre variabili!

2.1.5 Identità di Assegnamento: Lvalue vs Rvalue

Quando leggete un errore di compilazione che cita le parole `lvalue` o `rvalue`, fate riferimento a questa distinzione:

- **Lvalue (Left-value):** Identifica una locazione di memoria fisica persistente e modificabile (scrivibile). Deve stare a sinistra dell'operatore di assegnamento `=`. Esempio: il nome di una variabile (`x`).
- **Rvalue (Right-value):** Rappresenta un valore temporaneo, costante o calcolato che non possiede una locazione di memoria persistente a cui fare riferimento. Può stare solo a destra dell'operatore `=`. Esempi: un letterale numerico (`5`), il risultato di un calcolo (`x + 3`).

Cosa non potete fare:

```
1 5 = x;           // ERRORE: 5 ? un rvalue, non potete assegnargli una cella
   ↪ di memoria!
2 x + y = 10;      // ERRORE: l'espressione a sinistra non ? un lvalue
   ↪ scrivibile.
```

2.2 Input/Output e Formattazione

2.2.1 La Filosofia di `printf`

La funzione `printf` non serve solo a stampare stringhe statiche. Funziona come una stampante di modelli o template. Per stampare un numero a schermo, la CPU deve effettuare una conversione: deve prendere la rappresentazione binaria in RAM (es. il numero 5) e tradurla nel carattere testuale corrispondente (il codice ASCII del carattere '5').

Per farlo, forniamo a `printf` una **Stringa di Formato (Format String)** che funge da modello con dei segnaposto (specificatori di formato), seguita dall'elenco degli argomenti che andranno a riempire ordinatamente tali segnaposto:

```
1 printf("L'utente %s ha ottenuto %d punti.\n", "Alice", 150);
```

In questo esempio, `\{%s` (segnaposto per stringhe) riceve "Alice", mentre `\{%d` (segnaposto per interi decimali) riceve 150.

2.2.2 Specificatori di Formato Essenziali

Per questa prima fase del corso utilizzeremo quasi esclusivamente questi tre segnaposto fondamentali:

- `\{%d`: Stampa o legge un intero (`int`) in base 10 con segno.
- `\{%f`: Stampa un numero reale (`double` o `float`) in virgola fissa. Mostra di default 6 cifre decimali (es. 3.141593).

2.2.3 L'Input dall'Utente: La Funzione scanf

Mentre `printf` scrive verso l'esterno, la funzione `scanf` legge i dati digitati dall'utente sulla tastiera (dallo standard input `stdin`) e li memorizza nelle variabili del programma.

```
1 int age;
2 scanf("%d", &age);
```

Per utilizzare correttamente `scanf` dovete seguire tre regole ferree:

1. **Fornire il formato corretto:** `\{\}%d` per interi, `\{\}%lf` per i double (attenzione: mentre in `printf` potete usare `\{\}%f` per i double, in `scanf` dovete usare obbligatoriamente `\{\}%lf` - *long float* - altrimenti la memoria verrà corrotta scrivendo 4 byte al posto di 8. Parleremo meglio di questi specificatori in un'altra lezione.).
2. **Fornire l'indirizzo con l'operatore `\{\}&`:** `scanf` deve scrivere fisicamente nella memoria della variabile. Per farlo, ha bisogno dell'indirizzo fisico della cella RAM, ottenuto anteponendo l'operatore unario `\{\}&` ("indirizzo di") al nome della variabile. Vedremo questi dettagli meglio in seguito. Per ora, possiamo pensare a `\{\}&` come ad un operatore che ci permette di *accedere alla variabile in scrittura*, mentre di norma l'accesso sarebbe solo in lettura.
3. **Consumo del Whitespace:** `scanf` ignorerà automaticamente tutti gli spazi, tabulazioni e caratteri di invio (`\{\}\{\}\n`) iniziali fino a quando non incontrerà il primo carattere utile del tipo richiesto.

2.2.4 Esempio Pratico Completo: Calcolo dell'Area del Cerchio

Ecco un programma C autosufficiente che potete compilare per vedere l'I/O in azione:

```
1 #include <stdio.h>
2
3 int main(void) {
4     double raggio = 0.0;
5     double area = 0.0;
6     double PI = 3.1415926535;
7
8     printf("Inserisci il raggio del cerchio (numero reale): ");
9     scanf("%lf", &raggio); // Chiede il raggio all'utente
10
11     area = PI * raggio * raggio;
12
13     printf("\n--- Risultati ---\n");
14     printf("Raggio inserito: %f cm\n", raggio);
15     printf("Area calcolata: %f cm2\n", area);
16
17     return 0;
18 }
```

2.3 Espressioni, Operatori e Logica

2.3.1 Espressioni Aritmetiche, Precedenza ed Associatività

Gli operatori si dividono in:

- **Unari:** Lavorano su un solo operando (es. $-$ unario per invertire il segno, come in $-x$).
- **Binari:** Richiedono due operandi (es. $+$, $-$, $*$, $/$, $\{\}$, $\%$).

Per valutare espressioni complesse senza ambiguità, il compilatore C applica regole rigide di **Precedenza** (l'ordine di importanza degli operatori) e **Associatività** (la direzione in cui valutare gli operatori con la stessa priorità).

2.3.2 Ogni espressione restituisce un valore!

Un concetto cardine del C, spesso trascurato, è che **qualsiasi espressione (inclusa una singola istruzione di assegnamento) produce e restituisce sempre un valore di output**.

Ad esempio, l'espressione di assegnamento $x = 5$ non si limita a scrivere 5 nella memoria della variabile x , ma **restituisce essa stessa il valore 5 come output**. Questo consente ad esempio di concatenare gli assegnamenti come in $y = x = 5$ (che valuta $x = 5$ restituendo 5, che viene poi assegnato a y).

Ecco la tabella completa degli operatori principali con l'indicazione del loro valore restituito:

Tabella				
Priorità	Operatore	Descrizione	Valore Restituito in Output	Associatività
1 (Massima)	$+ - !$	Unari (Segno, NOT)	Il valore numerico invertito o lo stato invertito (1/0)	Da destra a sinistra
2 (Media)	$* / \{\} \%$	Moltiplicazione, Divisione, Modulo	Il risultato matematico dell'operazione	Da sinistra a destra
3 (Bassa)	$+ -$	Somma, Sottrazione	Il risultato matematico dell'operazione	Da sinistra a destra

4 (Minima)

=

Assegnamento
sempliceIl valore che
è stato
scritto nella
variabile a
sinistraDa destra a
sinistra

2.3.3 Esempio di Valutazione delle Espressioni

Consideriamo questa espressione complessa:

```
1 int x;
2 int y;
3 y = (x = 2 * 3) + 4;
```

Sotto il cofano, la CPU eseguirà le operazioni in questo preciso ordine rispettando le precedenze e le parentesi:

1. Risolve l'espressione tra parentesi tonde più interna: $x = 2 * 3$.
2. Esegue la moltiplicazione $2 * 3$ che restituisce 6.
3. Esegue l'assegnamento $x = 6$. Questa espressione scrive 6 in x e **restituisce in output 6**.
4. Sostituisce l'intera parentesi $(x = 2 * 3)$ con il suo valore di output 6. L'espressione ora diventa: $y = 6 + 4$.
5. Esegue la somma $6 + 4$ che restituisce 10.
6. Esegue l'assegnamento finale scrivendo 10 in y .

Suggerimento

Uso delle parentesi Utilizzate sempre le parentesi tonde () per forzare l'ordine desiderato di valutazione, chiarire il flusso di output ed evitare di scrivere espressioni ambigue o difficili da leggere.

2.3.4 Assegnamenti Composti

Per scrivere codice più compatto ed efficiente, il C offre degli operatori di assegnamento composto:

- $x += 5$; equivale a scrivere $x = x + 5$;
- $x *= 2$; equivale a scrivere $x = x * 2$;
- $x -= 1$; equivale a scrivere $x = x - 1$;

2.3.5 Il Trabocchetto della Divisione: Intera vs Reale

In C, il comportamento dell'operatore di divisione `/` è determinato unicamente dal tipo degli operandi:

- **Divisione Intera:** Se entrambi gli operandi sono interi (`int`), C esegue un calcolo intero e **tronca completamente la parte decimale** senza effettuare alcun arrotondamento. Esempio: `5 / 2` restituisce 2 (la frazione `.5` viene persa). Possiamo vedere questa divisione anche come **il quoziente** tra due numeri.
- **Divisione Reale:** Se almeno uno degli operandi è un numero reale (ad esempio `double`), C esegue una divisione decimale completa. Esempio: `5.0 / 2` promuove implicitamente l'intero 2 a `2.0` ed esegue `5.0 / 2.0`, restituendo `2.5`.

L'operatore **Modulo** (`\{\}%`) restituisce invece il resto della divisione intera e può essere utilizzato **solo tra interi**. Esempio: `5 \{\}% 2` restituisce 1. Viene comunemente usato per verificare se un numero è pari (`n \{\}% 2 == 0`).

2.4 Logica Booleana e Decisioni Condizionali

2.4.1 Come il C rappresenta la Verità

Nello standard C originale non esiste un tipo booleano puro nativo (sebbene standard moderni abbiano introdotto estensioni). Il C rappresenta i valori logici tramite interi:

- `0` rappresenta **Falso (False)**.
- **Qualsiasi valore diverso da 0** (solitamente 1) rappresenta **Vero (True)**.

Gli operatori di confronto (relazionali) permettono di stabilire relazioni d'ordine o di uguaglianza tra espressioni aritmetiche, producendo come output 1 se la relazione è verificata (Vero) e 0 in caso contrario (Falso):

Tabella

Operatore	Significato	Esempio	Risultato (se <code>a=5, b=10</code>)	Nota di Attenzione
<code>==</code>	Uguale a	<code>a == b</code>	0 (Falso)	Da non confondere con l'assegnamento <code>=!</code>

!=	Diverso da	a != b	1 (Vero)	Restituisce vero se i valori non coincidono
>	Maggiore di	a > b	0 (Falso)	Confronto stretto
<	Minore di	a < b	1 (Vero)	Confronto stretto
>=	Maggiore o uguale a	b >= 10	1 (Vero)	Vero se maggiore o esattamente identico
<=	Minore o uguale a	a <= 5	1 (Vero)	Vero se minore o esattamente identico

2.4.2 Operatori Logici Booleani

Per combinare più relazioni e creare espressioni logiche complesse, si utilizzano gli operatori logici:

Tabella				
Operatore	Nome Logico	Descrizione	Sintassi C	Regola di Output
&	AND (Congiunzione)	Entrambe vere	A & B	Restituisce 1 solo se sia A che B sono diversi da zero.
	OR (Disgiunzione)	Almeno una vera	A B	Restituisce 1 se almeno uno tra A e B è diverso da zero.
!	NOT (Negazione)	Inversione logica	!A	Restituisce 1 se A vale zero, e 0 se A è diverso da zero.

Tabelle di Verità (Truth Tables)

Per comprendere appieno l'output prodotto dagli operatori logici, si fa riferimento alle tabelle di verità (dove 0 è Falso e 1 è Vero):

Tabella

Operando A	Operando B	Risultato A		
		$\text{A} \& \text{B}$	$\text{A} \vee \text{B}$	!A
0 (Falso)	0 (Falso)	0	0	1
0 (Falso)	1 (Vero)	0	1	1
1 (Vero)	0 (Falso)	0	1	0
1 (Vero)	1 (Vero)	1	1	0

2.4.3 L'Istruzione Condizionale `if`

Per deviare il flusso lineare del programma a seconda delle decisioni logiche, utilizziamo il costrutto `if`:

```
1 if (condizione) {
2     // Questo blocco viene eseguito solo se la condizione restituisce un
   ↪ valore Vero (diverso da 0)
3 }
```

Ecco un esempio pratico in cui usiamo variabili intere per memorizzare stati logici:

```
1 #include <stdio.h>
2
3 int main(void) {
4     int haPatente = 1; // 1 rappresenta Vero, 0 rappresenta Falso
5     int eta = 20;
6
7     // L'operatore logico && combina i due controlli
8     if (eta >= 18 && haPatente) {
9         printf("Autorizzato alla guida dell'autovettura.\n");
10    }
11
12    // Il NOT logico inverte la condizione
13    int pioggia = 1; // 1 rappresenta Vero (Piove)
14    if (!pioggia) {
15        printf("Puoi fare una passeggiata all'aperto.\n");
16    } else {
17        printf("Prendi l'ombrello!\n");
18    }
19
20    return 0;
21 }
```

2.4.4 La Logica Cortocircuitata (Short-Circuit Evaluation)

Il compilatore C valuta le espressioni logiche composte da sinistra a destra ed **arresta immediatamente la valutazione non appena il risultato finale è matematicamente certo**:

- **Cortocircuito dell'AND ($\&$):** Nell'espressione $A \& B$, se A è Falso (0), l'intera espressione sarà sicuramente falsa indipendentemente da B . Pertanto, **B non viene valutato**.
- **Cortocircuito dell'OR ($\|$):** Nell'espressione $A \| B$, se A è Vero (1), l'intera espressione sarà sicuramente vera indipendentemente da B . Pertanto, **B non viene valutato**.

Questo comportamento pigro è di vitale importanza pratica per proteggere il programma da crash a runtime. Permette infatti di effettuare un controllo di sicurezza preliminare prima di eseguire operazioni potenzialmente fatali:

```
1 int divisore = 0;
2 int dividendo = 10;
3
4 // Grazie al cortocircuito, poich? "divisore != 0" ? Falso (0),
5 // la divisione per zero non viene eseguita e il programma non crasha.
6 if (divisore != 0 && (dividendo / divisore) > 2) {
7     printf("Condizione superata in sicurezza!\n");
8 }
```

Lo stesso principio si applica per evitare errori usando l'operatore **OR ($\|$)**: se la prima condizione è Vera (1), l'intera espressione è sicuramente vera, quindi la seconda parte non viene valutata. Questo è utile ad esempio per impostare un bypass o una scorciatoia di sicurezza:

```
1 int bypassaControlli = 1; // 1 rappresenta Vero
2 int divisore = 0;
3 int dividendo = 10;
4
5 // Grazie al cortocircuito, poich? "bypassaControlli" ? Vero (1),
6 // l'intera condizione dell'if ? gi? soddisfatta.
7 // La CPU non esegue la seconda parte, evitando la divisione per zero e
8 //   ↳ il crash!
9 if (bypassaControlli \| (dividendo / divisore) > 2) {
10     printf("Accesso consentito (controlli bypassati o superati).\n");
11 }
```

Se invertissimo l'ordine della condizione (`if ((dividendo / divisore) > 2 \| bypassaControlli)`), la CPU proverebbe immediatamente ad eseguire la divisione per zero, provocando l'arresto immediato del programma con un errore di tipo Floating point exception.

2.5 Quiz di Autovalutazione

Mettetevi alla prova con questo test rapido di 10 domande per verificare la vostra comprensione dei concetti di questa lezione. Cliccate su ciascuna domanda per verificare la risposta corretta e la spiegazione.

Quiz 1

Domanda

1. Sotto il cofano del calcolatore, che cos'è fisicamente una variabile?

- A) Una lettera algebrica salvata nel compilatore.
- B) Una denominazione simbolica associata a un indirizzo e ad un'area fisica della memoria RAM.
- C) Un file temporaneo memorizzato sull'hard disk.
- D) Un'istruzione in linguaggio macchina per eseguire una somma.

Soluzione
Risposta corretta: B
Spiegazione: Le variabili sono astrazioni create per lo sviluppatore umano; durante la compilazione vengono tradotte direttamente in indirizzi fisici o scostamenti della memoria RAM (solitamente nello Stack).

Quiz 2

Domanda

2. Quale delle seguenti affermazioni su una variabile locale non inizializzata è corretta?

- A) C assegna automaticamente il valore 0 ad essa.
- B) La compilazione fallisce immediatamente.
- C) Contiene un valore spazzatura (garbage value) indeterminato dovuto allo stato precedente dei bit in RAM.
- D) Viene eliminata automaticamente dal sistema operativo all'avvio.

Soluzione
Risposta corretta: C
Spiegazione: La dichiarazione riserva l'area di memoria ma non la azzerava. Leggere una variabile non inizializzata produce comportamenti imprecisati basati sui bit residui in quella cella.

Quiz 3

Domanda

3. Perché il linguaggio C richiede la dichiarazione esplicita del tipo per ogni variabile?

- A) Per consentire al compilatore di allocare la corretta dimensione in byte e generare istruzioni CPU specifiche ed efficienti.
- B) Per rendere il codice leggibile solo da programmatori esperti.

- C) Perché la memoria RAM gestisce solo tipi interi.
- D) Per consentire al linker di unire i file sorgente in un unico file eseguibile.

Soluzione
Risposta corretta: A
Spiegazione: Conoscere il tipo permette al compilatore di allocare lo spazio esatto (es. 4 byte per `int`, 8 byte per `double`) e di usare i registri e le istruzioni CPU appropriate (somma intera vs somma reale).

Quiz 4

Domanda

4. Nell'istruzione `x = y + 3;`, quali elementi rappresentano rispettivamente un lvalue e un rvalue?

- A) `x` è un lvalue; `y + 3` è un rvalue.
- B) `x` è un rvalue; `y + 3` è un lvalue.
- C) Entrambi sono lvalue perché contengono variabili.
- D) Entrambi sono rvalue perché fanno parte di un calcolo aritmetico.

Soluzione
Risposta corretta: A
Spiegazione: Un lvalue indica una locazione di memoria persistente in cui è possibile scrivere (la variabile `x`), mentre un rvalue è un valore calcolato temporaneo (`y + 3`) che non ha un indirizzo persistente accessibile.

Quiz 5

Domanda

5. Cosa succede se si usa lo specificatore `\{%f` all'interno della funzione `scanf` per leggere un numero di tipo `double`?

- A) Il numero viene letto correttamente ma approssimato all'intero più vicino.
- B) Si verifica una corruzione di memoria, poiché `scanf` si aspetta `\{%lf` per scrivere 8 byte anziché 4.
- C) La compilazione fallisce con un errore di sintassi.
- D) La variabile viene allocata nello Heap invece che sullo Stack.

Risposta corretta: B
Spiegazione: In `scanf`, lo specificatore `%f` è destinato ai dati di tipo `float` (4 byte). Per memorizzare correttamente un `double` (8 byte) è obbligatorio usare `%lf`.

Soluzione

Quiz 6

Domanda

6. Quanto restituisce l'espressione aritmetica `5 / 2` in C?

- A) 2.5
- B) 2
- C) 3
- D) 0

Risposta corretta: B
Spiegazione: Poiché entrambi gli operandi (5 e 2) sono interi (`int`), C esegue una divisione intera e tronca completamente la parte frazionaria.

Soluzione

Quiz 7

Domanda

7. Come si può ottenere una divisione decimale corretta (che restituisca 2.5) a partire dai numeri 5 e 2?

- A) Scrivendo `5.0 / 2.0` o `5 / 2.0`.
- B) Utilizzando l'operatore modulo `%`.
- C) Memorizzando il risultato in una variabile `double` senza modificare i due operandi interi.
- D) Modificando il codice di ritorno del programma `main` con `return 2.5;`.

Risposta corretta: A
Spiegazione: Se almeno uno degli operandi è un letterale reale (5.0), l'altro intero viene promosso implicitamente a reale prima dell'operazione, forzando una divisione in virgola mobile.

Soluzione

Quiz 8

Domanda

8. In C, come viene rappresentato internamente il valore logico Falso (False)?

- A) Tramite la parola chiave `false` inclusa di default nel compilatore.
- B) Esclusivamente dal valore intero 0.
- C) Da qualsiasi valore minore di 0.
- D) Dalla presenza di un valore spazzatura (garbage value).

Soluzione
Risposta corretta: B
Spiegazione: Per il linguaggio C nativo, lo 0 logico rappresenta il Falso, mentre qualsiasi valore diverso da zero (positivo o negativo) è interpretato come Vero.

Quiz 9

Domanda

9. Nel frammento `if (a != 0 & b / a > 1)`, cosa succede se la variabile `a` è uguale a 0?

- A) Il programma crasha immediatamente a causa di una divisione per zero.
- B) La condizione restituisce Vero perché la seconda parte non è controllata.
- C) La seconda parte della condizione non viene valutata grazie alla logica di cortocircuito dell'operatore `&`.
- D) Il valore di `b` viene azzerato in automatico dalla CPU.

Soluzione
Risposta corretta: C
Spiegazione: Poiché il primo termine `a != 0` valuta a Falso (0), il risultato globale di un AND (`&`) è matematicamente certo di essere Falso. La CPU arresta la valutazione evitando il crash dovuto alla divisione `b / a`.

Quiz 10

Domanda

10. Qual è il comportamento dell'operatore logico di cortocircuito OR (`||`)?

- A) Valuta sempre entrambi gli operandi per sicurezza.
- B) Se il primo operando è Vero (1), non valuta il secondo perché l'espressione è sicuramente Vera.
- C) Inverte sempre il risultato finale.

- D) Restituisce errore se uno dei due operandi è uguale a zero.

Soluzione
Risposta corretta: B
Spiegazione: Per l'operatore `||` basta che un solo operando sia vero affinché l'espressione totale sia vera. Se il primo termine è non-nullo, la CPU ottimizza saltando la valutazione del secondo.

2.6 Esercizi Pratici

Mettete in pratica i concetti studiati provando a risolvere questi esercizi. Le soluzioni in codice C sono nascoste all'interno dei relativi pannelli.

Esercizio 10: Calcolo di Sicurezza del Logaritmo (Bozza

Domanda

Logica)

Il calcolo del logaritmo naturale di un numero reale x (in matematica espresso come $\ln(x)$) è definito solo per numeri strettamente maggiori di zero ($x > 0$). Scrivete un programma in C che acquisisca un numero `double x` da tastiera e, utilizzando gli operatori logici e il cortocircuito, memorizzi in una variabile `valido` il valore 1 se è possibile calcolare il logaritmo in sicurezza, 0 altrimenti. *Vincolo: non utilizzate il costrutto `if` o `else`.*

Soluzione

```

1  #include <stdio.h>
2
3  int main(void) {
4      double x = 0.0;
5
6      printf("Inserisci il valore reale x: ");
7      scanf("%lf", &x);
8
9      // L'espressione relazionale restituisce 1 se vero, 0 se falso.
10     // Viene valutata in sicurezza.
11     int valido = (x > 0.0);
12
13     printf("E' possibile calcolare il logaritmo? (1=Si, 0=No): %d\n",
14           ← " ", valido);
15     return 0;
16 }

```

Spiegazione dell'algoritmo: In C le espressioni di confronto (come $x > 0.0$) producono un valore numerico pari a 1 se la relazione d'ordine è verificata o 0 in caso contrario. Assegniamo questo valore direttamente alla variabile `valido`, senza usare blocchi condizionali.

LEZIONE 3

La Cassetta degli Attrezzi del C



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 3

La Cassetta degli Attrezzi del C

Nelle lezioni precedenti abbiamo costruito le fondamenta del C: le variabili `int` e `double`, i formati base di `printf` e `scanf`, gli operatori aritmetici e la logica condizionale con `if`. In questa lezione completiamo la nostra **cassetta degli attrezzi** fondamentale: impareremo a scegliere il tipo più adatto per ogni situazione grazie ai modificatori (`unsigned`, `short`, `long`), a formattare l'output con precisione chirurgica grazie al padding di `printf`, a nominare i valori fissi con `const` e `\{\}\#define`, a convertire dati tra tipi diversi in modo consapevole e ad accedere alle funzioni matematiche avanzate della libreria `<math.h>`. Chiuderemo con una demo sui limiti dei tipi numerici: overflow, underflow, `inf` e `NAN`.

3.1 Il Sistema dei Tipi: Modificatori e Limiti

3.1.1 Riepilogo Rapido

Ricordate che un tipo di dato definisce due cose: **quanta memoria** viene allocata e **come vengono interpretati i bit** al suo interno. Abbiamo già incontrato `int` (4 byte, numeri interi) e `double` (8 byte, numeri reali in doppia precisione). Ma il C mette a disposizione una famiglia più ampia di tipi interi.

3.1.2 Modificatori di tipo, il tipo `char` (ASCII) e la dipendenza dalla macchina

- **La dipendenza dalla macchina (Machine Dependency)** Una delle caratteristiche più importanti del C è che **la dimensione in byte dei tipi di dato non è fissa o universale**. Lo standard C definisce solo dei requisiti minimi e delle relazioni d'ordine (es. `sizeof(short) <= sizeof(int) <= sizeof(long)`). La dimensione effettiva dipende dall'architettura del processore (16, 32 o 64 bit) e dal compilatore (il sistema operativo e le sue convenzioni di ABI). Il tipo `char` è l'unico garantito occupare **sempre 1 byte (8 bit)** in qualsiasi sistema compatibile.
- **I modificatori degli interi** Permettono di adattare il tipo alle proprie esigenze senza sprecare memoria:
 - `short int` (o `short`): tipicamente 2 byte, valori da -32.768 a +32.767. Utile per piccoli numeri.
 - `int`: tipicamente 4 byte, valori da -2 miliardi a +2 miliardi.

- `long int` (o `long`): tipicamente 8 byte su sistemi moderni a 64 bit, per numeri molto grandi.
 - `unsigned int` / `unsigned long`: rimuovono il segno, consentendo di rappresentare solo valori positivi e raddoppiando il range positivo.
- **Il tipo `char` è un intero piccolo sotto la maschera!** Spesso si pensa al tipo `char` come a qualcosa di radicalmente diverso perché serve a memorizzare lettere o simboli (es. `'a'`, `'?'`). **Sotto il cofano del calcolatore, un `char` è semplicemente un numero intero a 8 bit (1 byte).** Può memorizzare valori numerici compresi tra -128 e $+127$ (se con segno) o da 0 a 255 (se `unsigned char`).

L'associazione tra il numero binario memorizzato in RAM e il carattere visualizzato a schermo è governata da una convenzione di codifica: la **tabella ASCII** (American Standard Code for Information Interchange). Ad esempio, memorizzare il carattere `'A'` significa scrivere in RAM il valore intero 65 (pari al binario `01000001`).

L'equivalenza in pratica:

```
1 #include <stdio.h>
2 int main(void) {
3     char c = 'A';
4     printf("Visualizzato come carattere: %c\n", c); // Stampa: A
5     printf("Visualizzato come intero:    %d\n", c); // Stampa: 65
6
7     // Possiamo fare operazioni matematiche sui caratteri!
8     c = c + 1;
9     printf("Nuovo carattere:           %c\n", c); // Stampa: B (66
10    ↪ in ASCII)
11    return 0;
12 }
```

- **La Cultura dello Sviluppatore: Consultare la tabella ASCII** Non bisogna imparare i codici a memoria. Su qualsiasi sistema Unix/macOS basta digitare nel terminale:

```
1 man ascii
```

In alternativa, si può consultare la risorsa web di riferimento su devdocs.io/c/language/ascii.

- **Tabella dei range tipici e dipendenza dall'implementazione** Ecco i range tipici su una macchina a 64 bit moderna:

Tabella

Tipo	Byte (tipici)	Intervallo di Valori	Specificatore printf
<code>char</code>	1	da -128 a $+127$	<code>\{%c</code> (carattere) o <code>\{%d</code> (intero)
<code>unsigned char</code>	1	da 0 a 255	<code>\{%d</code> o <code>\{%u</code>

short	2	da -32.768 a $+32.767$	\{\}%d
unsigned short	2	da 0 a 65.535	\{\}%u
int	4	da $-2.147.483.648$ a $+2.147.483.647$	\{\}%d
unsigned int	4	da 0 a 4.294.967.295	\{\}%u
long	8	da $\approx$ -9.22×10^{18} a $\approx$ $+9.22 \times 10^{18}$	\{\}%ld
unsigned long	8	da 0 a $\approx$ 1.84×10^{19}	\{\}%lu
float	4	da $\approx$ 1.17×10^{-38} a $\approx$ 3.40×10^{38}	\{\}%f (o \{\}%e / \{\}%g)
double	8	da $\approx 2.22 \times$ 10^{-308} a $\approx$ 1.80×10^{308}	\{\}%f (o \{\}%e / \{\}%g)

3.1.3 Scoprire le Dimensioni con sizeof

L'operatore `sizeof` restituisce a tempo di compilazione il numero di byte occupato da un tipo o da una variabile. È molto utile per verificare il comportamento del compilatore sulla propria macchina:

```

1  #include <stdio.h>
2
3  int main(void) {
4      printf("char:      %zu byte\n", sizeof(char));
5      printf("short:     %zu byte\n", sizeof(short));
6      printf("int:        %zu byte\n", sizeof(int));
7      printf("long:       %zu byte\n", sizeof(long));
8      printf("float:      %zu byte\n", sizeof(float));
9      printf("double:    %zu byte\n", sizeof(double));
10     return 0;
11 }
```

Nota

Il formato `\{%zu` per `sizeof` restituisce un valore di tipo `size_t` (un intero senza segno, sui sistemi a 64 bit grande quanto un `unsigned long`). Lo specificatore corretto per stamparlo è `\{%zu`, non `\{%d`.

3.1.4 Accedere ai Limiti Esatti: `<limits.h>` e `<float.h>`

Non è necessario ricordare i valori limite a memoria. La libreria standard mette a disposizione costanti predefinite che il compilatore calcola automaticamente per la piattaforma corrente:

```

1  #include <stdio.h>
2  #include <limits.h>
3  #include <float.h>
4
5  int main(void) {
6      printf("INT_MAX = %d\n", INT_MAX);    // +2.147.483.647
7      printf("INT_MIN = %d\n", INT_MIN);    // -2.147.483.648
8      printf("UINT_MAX = %u\n", UINT_MAX);  // +4.294.967.295
9      printf("LONG_MAX = %ld\n", LONG_MAX);
10     printf("DBL_MAX = %e\n", DBL_MAX);    // il pi? grande double
        ↪ positivo
11     printf("DBL_MIN = %e\n", DBL_MIN);    // il pi? piccolo double
        ↪ positivo normalizzato
12     return 0;
13 }
```

La competenza chiave non è memorizzare questi numeri: è sapere *dove cercarli*. Aprire `<limits.h>` dalla documentazione di sistema è una delle prime abitudini da coltivare.

3.2 Codifica IEEE 754: Perché $0.1 + 0.2 \neq 0.3$

Un `double` ha 64 bit. Lo standard **IEEE 754** li divide così:

Tabella

Campo	Bit	Ruolo
Segno	1	Indica positivo/negativo
Esponente	11	Scala il numero (potenza di 2), codificato con <i>bias</i> 1023
Mantissa	52	Le cifre significative

Il valore rappresentato è:

$$(-1)^s \times 1.\text{mantissa} \times 2^{(e-1023)}$$

È la notazione scientifica, ma in **base 2**.

3.2.1 Perché esiste il bias 1023?

L'esponente reale può essere **negativo** (numeri molto piccoli) o **positivo** (numeri molto grandi).

Gli 11 bit dell'esponente possono rappresentare valori da 0 a 2047. Per rappresentare esponenti negativi, il *bias* 1023 "centra" la gamma:

- $e_{\text{mem}} = 0 \rightarrow e_{\text{reale}} = -1023$ (numeri minuscoli)
- $e_{\text{mem}} = 2047 \rightarrow e_{\text{reale}} = +1024$ (numeri enormi)

Così un double può rappresentare valori che vanno da circa (10^{-308}) a (10^{308}) .

3.2.2 Esempio: rappresentazione di 0.1

0.1 in binario è infinito:

$$0.1 = 0.00011001100110011001100 \dots_2$$

Normalizzato:

$$0.1 = 1.1001100110011 \dots \times 2^{-4}$$

Quindi:

- segno = 0
- esponente reale = -4 (mi sono "spostato" di 4 posizioni dopo la virgola)
- esponente memorizzato = $-4 + 1023 = 1019$
- mantissa = primi 52 bit dell'espansione infinita

3.2.3 Tabella dei campi del double per 0.1

Tabella			
Campo	Bit (valore binario)	Significato	Valore decimale
Segno	0	Numero positivo	+
Esponente	01111111011 (1019)	Esponente memorizzato	Esponente reale = $1019 - 1023 = -4$

3.3 Formati di Stampa printf — La Guida Completa

3.3.1 Tutti i Formati Essenziali

In L2 abbiamo usato principalmente `\{%d` e `\{%f`. Ecco il quadro completo dei formati che userete più spesso in questo corso:

Tabella

Formato	Tipo C	Esempio di output
<code>\{%d</code>	int (con segno)	42, -7
<code>\{%u</code>	unsigned int	4294967295
<code>\{%ld</code>	long int	9223372036854775807
<code>\{%lu</code>	unsigned long	18446744073709551615
<code>\{%zu</code>	size_t (risultato di sizeof)	8
<code>\{%f</code>	double (virgola fissa, 6 decimali default)	3.141593
<code>\{%e</code>	double (notazione scientifica)	3.141593e+00
<code>\{%g</code>	double (sceglie il formato più compatto)	3.14159
<code>\{%c</code>	char	A
<code>\{%s</code>	stringa (char *)	hello
<code>\{%\{%</code>	carattere letterale <code>\{%</code>	<code>\{%</code>

3.3.2 Controllo della Precisione Decimale

In L2 abbiamo visto `\{%f` con il formato di default (6 decimali). Possiamo controllare esattamente quante cifre mostrare usando il modificatore `.<N>` tra `\{%` e la lettera di formato:

- `\{%.2f` -> 2 cifre dopo la virgola (es. 3.14)
- `\{%.0f` -> solo la parte intera, senza virgola (es. 3)
- `\{%.3e` -> notazione scientifica con 3 cifre decimali (es. 3.142e+00)
- `\{%.17f` -> 17 cifre: utile per esporre gli errori di arrotondamento IEEE 754

```

1  #include <stdio.h>
2
3  int main(void) {
4      double pi = 3.14159265358979;
5      printf("Default:   %f\n", pi); // 3.141593
6      printf("2 decimali: %.2f\n", pi); // 3.14
7      printf("0 decimali: %.0f\n", pi); // 3
8      printf("Scientifica: %.3e\n", pi); // 3.142e+00
9      return 0;
10 }
```

3.3.3 Ampiezza Minima e Padding

Quando dobbiamo stampare dati in forma tabellare allineata, possiamo specificare l'**ampiezza minima** del campo aggiungendo un numero intero tra `\{ }%` e la lettera del formato:

Tabella

Specificatore	Effetto
<code>\{ }%10d</code>	Occupi almeno 10 caratteri, allineato a destra (spazi a sinistra)
<code>\{ }%-10d</code>	Allineato a sinistra (spazi a destra)
<code>\{ }%010d</code>	Padding con zeri invece di spazi
<code>\{ }%10.2f</code>	Almeno 10 caratteri totali, di cui 2 decimali

Esempio pratico – generare una tabella ben allineata:

```

1  #include <stdio.h>
2
3  int main(void) {
4      printf("%-20s %8s %10s\n", "Prodotto", "Quantit?", "Prezzo");
5      printf("%-20s %8d %10.2f\n", "Mela", 12, 0.89);
6      printf("%-20s %8d %10.2f\n", "Cioccolato", 3, 2.50);
7      printf("%-20s %8d %10.2f\n", "Acqua minerale", 6, 0.45);
8      return 0;
9  }
```

Output:

```

1  Prodotto           Quantit?      Prezzo
2  Mela                12          0.89
3  Cioccolato          3           2.50
4  Acqua minerale      6           0.45
```

Suggerimento

Documentazione completa di `printf` Non è necessario memorizzare tutti i flag. Consultate man 3 `printf` nel terminale oppure devdocs.io/c/io/fprintf per la documentazione interattiva completa.

3.4 Costanti: Tre Modi per Dare un Nome a un Valore Fisso

Immaginate di scrivere il valore 3.14159 cinque volte in uno stesso programma. Cosa succede se vi siete sbagliati di una cifra in un solo punto? Le costanti nominali risolvono questo problema.

3.4.1 Letterali Tipizzati

I valori che inserisco direttamente nelle istruzioni del programma vengono chiamati **letterali**. Un valore letterale come 75 o 3.14 ha un tipo implicito. Il compilatore lo determina automaticamente, ma a volte dobbiamo essere espliciti:

```
1 75          // int
2 75u         // unsigned int
3 75l         // long int
4 3.14        // double (attenzione: non float!)
5 3.14f       // float
6 3.14159L    // long double
```

Questo è importante quando si passano argomenti a funzioni che si aspettano un tipo specifico, o quando si sommano tipi diversi.

3.4.2 const – Variabile Immutabile

Il modo più diretto e moderno per definire una costante nominale è la keyword `const`:

```
1 const double PI = 3.14159265358979;
2 const int MAX_STUDENTI = 200;
```

Il compilatore garantisce che nessun'altra parte del codice possa modificare il valore. Se tentate di farlo, otterrete un errore di compilazione:

```
1 const double PI = 3.14;
2 PI = 3.1415; // ERRORE: assignment of read-only variable 'PI'
```

Nota

Regola fondamentale per `const` Dichiarazione e inizializzazione devono avvenire sulla stessa riga. `const int x;` seguito da `x = 5;` è un errore: una costante deve essere inizializzata al momento della sua definizione.

Quando usare `const`: ogni volta che un valore ha un significato semantico preciso e non deve mai cambiare nel programma (aliquote fiscali, costanti fisiche, dimensioni di array, etc.).

3.4.3 #define – Sostituzione del Preprocessore

Un'alternativa storica a `const` è la **macro del preprocessore**:

```
1 #define VOTO_MINIMO 18
2 #define CORSO      "Programmazione 1"
```

A differenza di `const`, `\{\}#define` non crea una variabile: il **preprocessore** sostituisce meccanicamente ogni occorrenza del nome con il testo corrispondente *prima ancora che il compilatore veda il codice*.

Possiamo osservare questo comportamento direttamente usando il flag `-E` di GCC, che ferma il processo dopo la fase di preprocessing e scrive il risultato in un file `.i`:

```
1  /* demo_define.c ? senza nessuna #include per vedere l'output pulito */
2  #define VOTO_MINIMO  18
3
4  int main(void) {
5      return VOTO_MINIMO;
6  }
```

```
1  gcc -E demo_define.c -o demo_define.i
2  cat demo_define.i
```

Output (semplificato):

```
1  # "demo_define.c"
2  # "<built-in>" 1
3  ...
4
5  int main(void) {
6      return 18;
7  }
```

Il nome `VOTO\_{}\_MINIMO` è completamente scomparso: il compilatore vedrà solo 18. Per convenzione i nomi di macro si scrivono in **MAIUSCOLO** per distinguerli visivamente dalle variabili.

3.4.4 Costanti Predefinite dalla Libreria Standard

La libreria standard offre già molte costanti pronte all'uso, definite tramite `\{}\#define`:

- `M\{}\_PI` (in `<math.h>`) -> pi con massima precisione da double
- `M\{}\_E` (in `<math.h>`) -> il numero di Eulero *e*
- `INT\{}\_MAX`, `INT\{}\_MIN` (in `<limits.h>`) -> limiti degli interi
- `DBL\{}\_MAX`, `DBL\{}\_EPSILON` (in `<float.h>`) -> limiti e precisione dei double

3.5 Conversioni di Tipo: Implicite ed Esplicite

3.5.1 Conversione Implicita (Promozione)

Quando un'espressione mescola tipi diversi, il C promuove automaticamente al tipo "più capiente" per evitare la perdita di informazioni:

- `int + double` -> il compilatore converte l'`int` in `double` prima di sommare
- `double` assegnato a `int` -> la parte decimale viene **troncata** (non arrotondata!)

```

1 double x = 7.9;
2 int y = x;           // y vale 7, non 8! Troncamento silenzioso.
3 printf("%d\n", y);   // stampa: 7

```

Questo troncamento silenzioso è una fonte comune di bug. Attenzione: né `-Wall` né `-Wextra` lo segnalano; con gcc serve il flag `-Wconversion` (warning: conversion from 'double' to 'int' may change value). Se il warning compare, non ignoratelo.

3.5.2 Cast Esplicito

Per convertire un tipo in modo deliberato e controllato, si usa il **cast esplicito**: si scrive il tipo di destinazione tra parentesi tonde prima dell'espressione.

```

1 int a = 5, b = 2;
2
3 double risultato = (double)a / b;   // 2.5 ? cast prima della divisione
4 double sbagliato = (double)(a / b); // 2.0 ? divisione intera, poi cast
5
6 printf("%.1f vs %.1f\n", risultato, sbagliato);

```

Attenzione

L'ordine del cast è fondamentale `(double)a / b` e `(double)(a / b)` producono risultati diversi! Nel primo caso il cast avviene prima della divisione, quindi il compilatore esegue una divisione reale. Nel secondo, la divisione intera è già avvenuta (perdendo il resto) quando il cast viene applicato.

3.5.3 Arrotondamento con `<math.h>`: `round`, `floor`, `ceil`

Il troncamento implicito di `(int)x` non è sempre il comportamento desiderato. La libreria `<math.h>` offre tre funzioni di arrotondamento con semantiche diverse:

- `round(x)` -> arrotondamento tradizionale: 2.4 -> 2.0, **2.5 -> 3.0**
- `floor(x)` -> sempre verso il basso (*pavimento*): 2.9 -> 2.0, **-2.1 -> -3.0**
- `ceil(x)` -> sempre verso l'alto (*soffitto*): 2.1 -> 3.0, **-2.9 -> -2.0**

Tutte e tre restituiscono un `double`. Per ottenere un `int`, combinate con il cast:

```

1 #include <math.h>
2
3 int n = (int)round(7.6);   // n = 8
4 int m = (int)floor(7.6);   // m = 7
5 int p = (int)ceil(7.2);    // p = 8

```

3.6 La Libreria <math.h>

3.6.1 Perché una Libreria Separata?

Le funzioni trigonometriche, le potenze e le radici richiedono istruzioni macchina specializzate (FPU — Floating Point Unit). Il C non le include nel nucleo del linguaggio: si importano esplicitamente con `\{\}\#include <math.h>`.

Nota

Compilare con `-lm` su Linux/GCC Su sistemi Linux con GCC, la libreria matematica deve essere linkata esplicitamente aggiungendo `-lm` al comando di compilazione: `gcc programma.c -lm -o programma`. Su macOS e con Clang questo non è necessario.

3.6.2 Le Funzioni Principali

Tabella

Funzione	Descrizione	Esempio
<code>sqrt(x)</code>	Radice quadrata	<code>sqrt(9.0) -> 3.0</code>
<code>pow(x, y)</code>	x^y	<code>pow(2, 10) -> 1024.0</code>
<code>fabs(x)</code>	Valore assoluto per <code>double</code>	<code>fabs(-3.5) -> 3.5</code>
<code>sin(x), cos(x), tan(x)</code>	Funzioni trigonometriche (in radianti)	<code>sin(M\{\}_PI) ~ 0.0</code>
<code>log(x)</code>	Logaritmo naturale ($\ln$)	<code>log(M\{\}_E) -> 1.0</code>
<code>log10(x)</code>	Logaritmo in base 10	<code>log10(100.0) -> 2.0</code>
<code>round(x), floor(x), ceil(x)</code>	Arrotondamenti (vedi §5)	

La costante `M\{\}_PI` definisce pi con la massima precisione disponibile nel tipo `double`.

3.6.3 Demo Pratica: Calcolatore di Moto Uniformemente Accelerato

Il seguente programma utilizza tutto ciò che abbiamo visto in questa lezione: tipi, formati con padding, costanti, <math.h> e casting:

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void) {
5      double v0, a, t;
6
7      printf("Velocit? iniziale (m/s): ");
8      scanf("%lf", &v0);

```



```

9   printf("Accelerazione (m/s2):  ");
10  scanf("%lf", &a);
11  printf("Tempo (s):                ");
12  scanf("%lf", &t);
13
14  double s = v0 * t + 0.5 * a * t * t; // spazio percorso
15  double vf = v0 + a * t;              // velocit? finale
16  double ec = 0.5 * vf * vf;           // energia cinetica
    ↪ specifica (massa=1 kg)
17
18  printf("\n--- Risultati ---\n");
19  printf("Spazio percorso:      %10.3f m\n", s);
20  printf("Velocit? finale:      %10.3f m/s\n", vf);
21  printf("Energia cinetica:      %10.3f J/kg\n", ec);
22
23  return 0;
24 }

```

Suggerimento

Dove trovare la documentazione completa di `<math.h>` devdocs.io/c/numeric/math elenca tutte le funzioni con firma, descrizione ed esempi.

3.7 Overflow, Underflow, inf e NAN

3.7.1 Cosa Succede Quando Superiamo i Limiti degli Interi?

Un `int` a 32 bit ha un valore massimo di `INT_MAX` = 2.147.483.647. Cosa succede aggiungendo 1?

```

1  #include <stdio.h>
2  #include <limits.h>
3
4  int main(void) {
5      int x = INT_MAX;
6      printf("INT_MAX      = %d\n", x);
7      printf("INT_MAX + 1 = %d\n", x + 1); // overflow!
8      return 0;
9  }

```

Output:

```

1  INT_MAX      = 2147483647
2  INT_MAX + 1 = -2147483648

```

Il risultato è **negativo**: siamo tornati al valore minimo rappresentabile, come un orologio che dopo le 23:59 torna alle 00:00. Questo fenomeno si chiama **overflow**. Attenzione: qui il compilatore **non** emette alcun warning, nemmeno con `-Wall -Wextra`, perché il valore di `x` è noto solo durante l'esecuzione. `gcc` avvisa (`-Woverflow`, attivo di default) solo quando l'overflow avviene in un'espressione costante calcolata in compilazione, come `INT_MAX`

+ 1 scritto direttamente. Per scoprire l'overflow durante l'esecuzione si può compilare con `-fsanitize=undefined`, che stampa runtime error: signed integer overflow.

Overflow degli interi: comportamento indefinito

Tecnicamente, in C l'overflow degli interi con segno produce **comportamento indefinito** (Undefined Behavior). In pratica produce il wraparound che vedete, ma non è garantito dallo standard. Per somme su grandi valori, promuovete prima a long: `(long)x + 1`.

3.7.2 Overflow e Valori Speciali nei Numeri in Virgola Mobile

I double si comportano diversamente dagli interi: invece di "wrapare", producono **valori speciali** definiti dallo standard IEEE 754:

```
1 #include <stdio.h>
2
3 int main(void) {
4     float troppo_grande = 10e10f * 10e30f;
5     printf("%f\n", troppo_grande); // inf
6     printf("%f\n", 0.0 / 0.0);    // nan
7     printf("%f\n", 1.0 / 0.0);    // inf
8     printf("%f\n", -1.0 / 0.0);   // -inf
9     return 0;
10 }
```

- **inf**: il risultato è troppo grande per essere rappresentato (overflow floating point).
- **nan** (*Not A Number*): risultato di un'operazione matematicamente indefinita ($0.0/0.0$, $\text{sqrt}(-1.0)$, etc.).
- **-inf**: infinito negativo.

La caratteristica più pericolosa di questi valori è che si **propagano**: qualsiasi operazione che coinvolge nan produce nan. Un nan che entra in un calcolo complesso può silenziosamente corrompere tutti i risultati successivi senza generare errori.

3.7.3 La Regola d'Oro: Compilate Sempre con `-Wall -Wextra`

```
1 gcc -Wall -Wextra -o programma programma.c
```

Questi flag attivano i principali avvisi del compilatore (ad esempio formati `printf` errati, variabili inutilizzate o non inizializzate). Non coprono tutto: le conversioni implicite con perdita di dati richiedono `-Wconversion` e l'overflow su variabili si scopre solo in esecuzione (`-fsanitize=undefined`). Un programma che compila con warnings non è un programma corretto.

3.8 Quiz di Autovalutazione

Mettetevi alla prova con queste 10 domande per verificare la comprensione dei concetti della lezione. Cliccate su ogni domanda per scoprire la risposta corretta e la spiegazione.

Quiz 1

Domanda

1. Qual è la differenza fondamentale tra `int` e `unsigned int` in termini di rappresentazione in memoria?

- A) `unsigned int` occupa il doppio dei byte di `int`.
- B) `int` e `unsigned int` occupano lo stesso numero di byte; cambia solo l'interpretazione del bit più significativo.
- C) `unsigned int` non può essere usato in espressioni aritmetiche.
- D) `int` è più preciso di `unsigned int` nei calcoli in virgola mobile.

Soluzione
Risposta corretta: B
Spiegazione: Entrambi i tipi occupano tipicamente 4 byte (32 bit). In `int`, il bit più significativo è il bit del segno (0=positivo, 1=negativo) che dimezza il range positivo. In `unsigned int`, quel bit contribuisce al valore numerico, portando il massimo da ~2 miliardi a ~4 miliardi.

Quiz 2

Domanda

2. Quale specificatore di formato va usato in `printf` per stampare correttamente un valore di tipo `long int`?

- A) `\{%d`
- B) `\{%ld`
- C) `\{%li`
- D) Sia `\{%ld` che `\{%li` sono corretti.

Soluzione
Risposta corretta: D
Spiegazione: Per `long int`, sia `\{%ld` (long decimal) che `\{%li` (long integer) sono corretti e producono lo stesso output. La scelta convenzionale è `\{%ld`.

Quiz 3

Domanda

3. Cosa produce il codice `printf("\{%%.17f\}\}\n", 0.1 + 0.2);` in C?

- A) `0.300000000000000000`
- B) Un errore di compilazione perché la precisione richiesta è troppo alta.
- C) `0.300000000000000004` (o un valore simile diverso da 0.3 esatto).

- D) 0.30000000 (7 cifre decimali, il massimo per double).

Soluzione
Risposta corretta: C
Spiegazione: Per lo standard IEEE 754, né 0.1 né 0.2 si rappresentano esattamente in base 2. La loro somma introduce un errore di arrotondamento che diventa visibile aumentando la precisione di stampa. Questo è il motivo fondamentale per cui non si devono confrontare due double con ==.

Quiz 4

Domanda

4. Cosa stampa il seguente codice?

```
1 int a = 5, b = 2;
2 double r = (double)(a / b);
3 printf("%.1f\n", r);
```

- A) 2.5
- B) 2.0
- C) 0.0
- D) Un errore di compilazione.

Soluzione
Risposta corretta: B
Spiegazione: Il cast (double) viene applicato dopo la divisione. Poiché sia a che b sono int, la divisione a / b è una divisione intera che produce 2 (troncando il resto). Solo successivamente il valore 2 viene convertito in 2.0. Per ottenere 2.5 bisogna scrivere (double)a / b.

Quiz 5

Domanda

5. Quale delle seguenti affermazioni su const è corretta?

- A) Una variabile const può essere modificata usando il cast (int).
- B) Una variabile const può essere dichiarata senza iniziarla e inizializzata in seguito.
- C) Il compilatore emette un errore se si tenta di modificare una variabile const.
- D) const è equivalente a \{\}\#define e produce lo stesso codice macchina.

Soluzione
Risposta corretta: C
Spiegazione: Il compilatore C garantisce l'immutabilità di una variabile `const` emettendo un errore di compilazione per qualsiasi tentativo di assegnamento successivo. A differenza di `\{\}\#define, const` ha un tipo e uno scope ben definiti, ed è soggetto al type checking del compilatore.

Quiz 6

Domanda

6. Il flag `-E` di GCC serve a:

- A) Attivare tutti i warning di errore (`-E` sta per "Errors enabled").
- B) Fermare la compilazione dopo la fase di preprocessing e produrre un file `.i`.
- C) Ottimizzare il codice generato per la massima efficienza.
- D) Abilitare le estensioni GNU non standard del linguaggio.

Soluzione
Risposta corretta: B
Spiegazione: Con `gcc -E sorgente.c -o sorgente.i`, GCC esegue solo il preprocessing (che risolve `\{\}\#include, \{\}\#define` e le direttive condizionali) e scrive il risultato nel file `.i`. Questo è utile per capire cosa "vede" effettivamente il compilatore dopo che le macro sono state espanse.

Quiz 7

Domanda

7. Cosa restituisce `floor(-2.1)` in C?

- A) `-2.0`
- B) `-3.0`
- C) `2.0`
- D) `-2` (di tipo `int`).

Soluzione
Risposta corretta: B
Spiegazione: `floor` restituisce il numero intero più grande che sia minore o uguale all'argomento (in italiano: arrotonda sempre verso il basso, cioè verso $-\infty$). Per `-2.1`, il numero intero più grande che è ancora ≤ -2.1 è `-3`. Nota: tutte le funzioni di arrotondamento di `<math.h>` restituiscono un `double`.

Quiz 8

Domanda

8. Quale dei seguenti formati `printf` stampa il numero 42 occupando esattamente 8 caratteri con zeri a sinistra?

- A) `\{\}%8d`
- B) `\{\}%-8d`
- C) `\{\}%08d`
- D) `\{\}%z8d`

Soluzione
Risposta corretta: C
Spiegazione: `\{\}%08d` specifica un'ampiezza minima di 8 caratteri con padding tra i caratteri (flag 0). Produce `00000042`. `\{\}%8d` produce 42 (spazi a sinistra). `\{\}%-8d` produce 42 (allineato a sinistra con spazi a destra).

Quiz 9

Domanda

9. Cosa stampa questo codice?

```
1 #include <stdio.h>
2 #include <limits.h>
3 int main(void) {
4     int x = INT_MAX;
5     printf("%d\n", x + 1);
6     return 0;
7 }
```

- A) 2147483648
- B) 0
- C) -2147483648
- D) Il programma non compila.

Soluzione
Risposta corretta: C
Spiegazione: Aggiungere 1 a `INT_MAX` causa un overflow: il valore "wrapa" e torna al valore minimo rappresentabile da un `int` a 32 bit, che è -2147483648. Il compilatore non emette alcun warning (nemmeno con `-Wall -Wextra`), perché l'overflow avviene su una variabile durante l'esecuzione. Tecnicamente si tratta di comportamento indefinito (undefined behavior), ma in pratica su architetture comuni si osserva questo comportamento.

Quiz 10

Domanda

10. Su Linux con GCC, perché è necessario aggiungere `-lm` al comando di compilazione quando si usa `<math.h>`?

- A) Per abilitare le estensioni GNU non standard.
- B) Per istruire il linker a includere la libreria matematica (`libm`) che contiene il codice delle funzioni come `sqrt` e `sin`.
- C) Perché `-lm` attiva la modalità "long math" che usa numeri a 128 bit.
- D) Non è necessario: `gcc` include automaticamente tutte le librerie standard.

Soluzione
Risposta corretta: B
Spiegazione: In C, la libreria matematica (`libm`) è separata dalla libreria standard C (`libc`). Il flag `-lm` dice al linker di collegare `libm` all'eseguibile finale, rendendo disponibili le implementazioni di `sqrt`, `sin`, `pow`, etc. Senza questo flag, il linker non troverà queste funzioni e la compilazione fallirà con errori "undefined reference". Su macOS e con Clang questo non è necessario perché `libm` è inclusa automaticamente.

3.9 Esercizi Pratici

Esercizio 1: Tabella dei Limiti

Domanda

Scrivete un programma che stampi una tabella formattata con tre colonne (TIPO, DIMENSIONE, VALORE MASSIMO) per i tipi `int`, `unsigned int`, `long` e `unsigned long`, usando `<limits.h>`. Assicuratevi che le colonne siano allineate correttamente usando i modificatori di ampiezza di `printf`.

Spiegazione: Usiamo `\t` per la colonna del tipo (allineata a sinistra, 20 caratteri), `\t` per la dimensione (8 caratteri, `\t` perché `sizeof` restituisce `size_t`) e l'ampiezza appropriata per il valore massimo.

```

1  #include <stdio.h>
2  #include <limits.h>
3
4  int main(void) {
5      printf("%-20s %8s %25s\n", "TIP0", "BYTE", "VALORE MASSIMO");
6      printf("%-20s %8zu %25d\n", "int", "int",
7             sizeof(int), "unsigned int", "unsigned
8             sizeof(long), "long", "long",
9             sizeof(long), "unsigned long", "unsigned
10             long", ULONG_MAX);
11     return 0;

```

Soluzione

Esercizio 2: Divisione con Cast

Domanda

Scrivete un programma che legga due interi `a` e `b` da tastiera e stampi: il risultato della divisione intera, il risultato della divisione reale (usando il cast), e il resto.

Soluzione

```

1  #include <stdio.h>
2
3  int main(void) {
4      int a, b;
5      printf("Inserisci il dividendo: ");
6      scanf("%d", &a);
7      printf("Inserisci il divisore: ");
8      scanf("%d", &b);
9
10     int quoziente = a / b;
11     double reale = (double)a / b;
12     int resto = a % b;
13
14     printf("Divisione intera: %d\n", quoziente);
15     printf("Divisione reale: %.4f\n", reale);
16     printf("Resto: %d\n", resto);
17
18     return 0;
19 }

```

Spiegazione: Il cast (double) a prima della divisione promuove l'operazione a divisione reale. L'operatore \{\}\% funziona solo tra interi e restituisce il resto della divisione intera.

Esercizio 3: Area e Perimetro con**Domanda**

$M\{\}_PI$

Scrivete un programma che legga il raggio di un cerchio e stampi area e circonferenza usando la costante $M\{\}_PI$ definita in `<math.h>`. Formattate l'output con 4 cifre decimali.

Spiegazione: `M_PI` è una macro predefinita in `<math.h>` che contiene il valore di π con la massima precisione del tipo `double`. Non è necessario definire manualmente `3.14159`.

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void) {
5      double raggio;
6      printf("Inserisci il raggio: ");
7      scanf("%lf", &raggio);
8
9      double area = M_PI * raggio * raggio;
10     double circonferenza = 2.0 * M_PI * raggio;
11
12     printf("Area: %.4f\n", area);
13     printf("Circonferenza: %.4f\n", circonferenza);
14
15     return 0;
16 }
```

Soluzione

Esercizio 4: Calcolatore di Ipotenusa

Domanda

Scrivete un programma che legga i due cateti di un triangolo rettangolo e calcoli l'ipotenusa usando `sqrt` e `pow`.

Spiegazione: Il teorema di Pitagora si implementa direttamente con `sqrt` e `pow`. Nota-
re che `a*a` è preferibile a `pow(a, 2)` per prestazioni (meno overhead), ma entrambe
le versioni sono corrette.

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void) {
5      double a, b;
6      printf("Cateo a: ");
7      scanf("%lf", &a);
8      printf("Cateo b: ");
9      scanf("%lf", &b);
10
11     double ipotenusa = sqrt(pow(a, 2) + pow(b, 2));
12     // oppure, equivalentemente: sqrt(a*a + b*b)
13     printf("Ipotenusa: %.4f\n", ipotenusa);
14     return 0;
15 }

```

Soluzione

Esercizio 5: Convertitore Angoli Radianti/Gradi

Domanda

Scrivete un programma che legga un angolo in gradi e stampi il suo seno, coseno e tangente, dopo averlo convertito in radianti (usando `M_PI`). Formattate l'output a 6 cifre decimali.

Spiegazione: La formula di conversione è $\text{rad} = \text{gradi} \times \pi / 180$. Le funzioni trigonometriche del C lavorano sempre in radianti.

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void) {
5      double gradi;
6      printf("Inserisci un angolo in gradi: ");
7      scanf("%lf", &gradi);
8
9      double radianti = gradi * M_PI / 180.0;
10
11     printf("Angolo in radianti: %.6f\n", radianti);
12     printf("Seno: %.6f\n", sin(radianti));
13     printf("Coseno: %.6f\n", cos(radianti));
14     printf("Tangente: %.6f\n", tan(radianti));
15
16     return 0;
17 }
```

Soluzione

Esercizio 6: Tabella di Arrotondamenti

Domanda

Scrivete un programma che legga un numero reale e stampi in una tabella i risultati di $(\text{int})x$ (troncamento), $\text{round}(x)$, $\text{floor}(x)$ e $\text{ceil}(x)$.

Spiegazione: Confrontare i quattro metodi sullo stesso valore (es. -2.7) evidenzia chiaramente la differenza tra troncamento, arrotondamento tradizionale, arrotondamento verso il basso e verso l'alto.

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main(void) {
5     double x;
6     printf("Inserisci un numero reale: ");
7     scanf("%lf", &x);
8
9     printf("\n%-20s %10.4f\n", "Valore originale:", x);
10    printf("\n%-20s %10d\n", "Troncamento (int):", (int)x);
11    printf("\n%-20s %10.4f\n", "round(x):", round(x));
12    printf("\n%-20s %10.4f\n", "floor(x):", floor(x));
13    printf("\n%-20s %10.4f\n", "ceil(x):", ceil(x));
14
15    return 0;
16 }
```

Soluzione

Esercizio 7: Demo Overflow

Domanda

Scrivete un programma che dimostri l'overflow degli interi: partite da `INT_MAX`, sommategli 1 e stampate il risultato. Poi risolvete il problema promuovendo l'operazione a `long`.

Spiegazione: `x + 1` produce un overflow perché l'addizione avviene tra due `int` (il letterale 1 è un `int`). `(long)x + 1` promuove prima `x` a `long`, poi la somma avviene in aritmetica a 64 bit e il risultato è corretto.

```

1 #include <stdio.h>
2 #include <limits.h>
3
4 int main(void) {
5     int x = INT_MAX;
6     printf("INT_MAX\n", x);
7     printf("INT_MAX + 1\n", x + 1);
8     // overflow:
9     printf("risultato negativo!\n", (long)x + 1);
10    printf("INT_MAX+1 = %ld\n", (long)x + 1);
11    // corretto:
12    printf("promozione a long\n", (long)x + 1);
13
14    return 0;
15 }
```

Soluzione

Esercizio 8: Cedolino Stipendio con

Domanda

const e Padding

Scrivete un programma che legga il reddito lordo mensile da tastiera e stampi un cedolino formattato che mostri: reddito lordo, imponibile IRPEF (23% della parte fino a 15.000€/anno), contributi INPS (9.19%), e netto mensile. Definite tutte le aliquote come costanti const.

Spiegazione: Le aliquote definite con const rendono il codice auto-documentante e sicuro. Il padding con \{\}\%-25s e \{\}\%10.2f produce un output tabellare allineato e

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void) {
5      const double ALIQUOTA_IRPEF = 0.23;
6      const double ALIQUOTA_INPS = 0.0919;
7
8      double lordo_mensile;
9      printf("Reddito lordo mensile (?) : ");
10     scanf("%lf", &lordo_mensile);
11
12     double inps = lordo_mensile * ALIQUOTA_INPS;
13     double irpef = lordo_mensile * ALIQUOTA_IRPEF;
14     double netto = lordo_mensile - inps - irpef;
15
16     printf("\n--- CEDOLINO ---\n");
17     printf("%10.2f %10.2f %10.2f %10.2f\n", "Lordo mensile:", "Contributi INPS:", "Irpef:", "Netto mensile:");
18     printf("%10.2f %10.2f %10.2f %10.2f\n", "Ritenuta IRPEF:", "Irpef:", "Netto mensile:", "Netto");
19     printf("%10.2f %10.2f %10.2f %10.2f\n", "Ritenuta IRPEF:", "Irpef:", "Netto mensile:", "Netto");
20     printf("%10.2f %10.2f %10.2f %10.2f\n", "Ritenuta IRPEF:", "Irpef:", "Netto mensile:", "Netto");
21     printf("%10.2f %10.2f %10.2f %10.2f\n", "Ritenuta IRPEF:", "Irpef:", "Netto mensile:", "Netto");
22     return 0;
23 }
```

Soluzione

Esercizio 9: Verifica Teorema di Pitagora con Tolleranza

Domanda

Scrivete un programma che legga tre numeri reali e verifichi se possono essere i lati di un triangolo rettangolo, usando fabs e una tolleranza di $1e-6$ per confrontare i double (senza usare if/else).

Spiegazione: Non si usa `==` per confrontare i `double` (errori di arrotondamento IEEE 754). Si calcola la differenza assoluta con `fabs` e si verifica che sia sotto la soglia di tolleranza `1e-6`. Il risultato dell'espressione `(differenza > 1e-6)` è 1 (Vero) o 0 (Falso), utilizzabile direttamente senza `if`.

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void) {
5      double a, b, c;
6      printf("Inserisci i tre lati: ");
7      scanf("%lf%lf%lf", &a, &b, &c);
8
9      // c deve essere il lato pi? lungo (ipotenusa candidata)
10     double ipotenusasq = c * c;
11     double cateti_sq = a * a + b * b;
12     double differenza = fabs(ipotenusasq - cateti_sq);
13
14     int e Rettangolo = (differenza < 1e-6);
15     printf("? un triangolo rettangolo (con c come ipotenusa): %d\n",
16           e_Rettangolo);
17     return 0;
18 }
```

Soluzione

Esercizio 10: Notazione Scientifica e Formati

Domanda

Scrivete un programma che stampi il numero di Avogadro (6.022×10^{23}) e la costante di Planck (6.626×10^{-34}) in tre formati diversi: `\{ }\%f`, `\{ }\%e`, e `\{ }\%g`. Osservate le differenze nell'output.

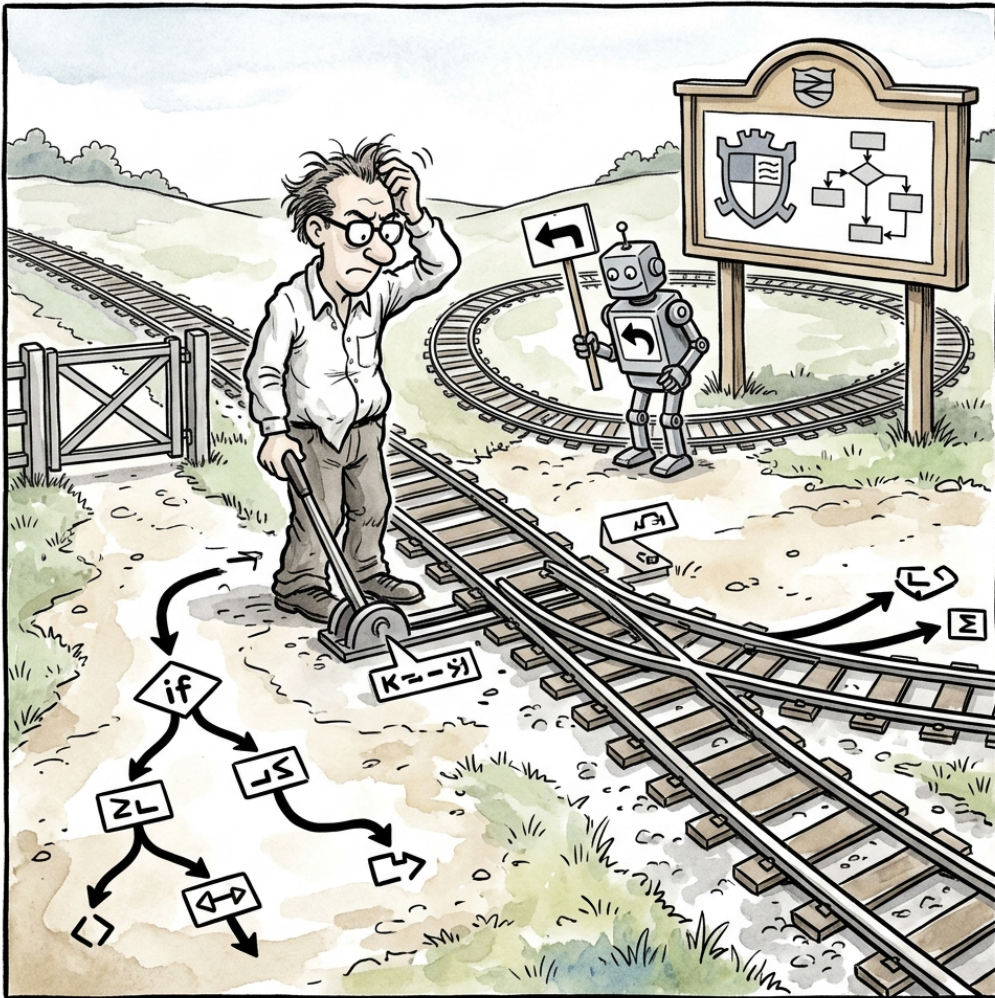
Spiegazione: \{\} tenta di stampare tutti i decimali (inutile per numeri molto grandi o piccoli), \{\}e forza sempre la notazione scientifica, \{\}g sceglie automaticamente il formato più compatto e leggibile. Per costanti fisiche, \{\}e o \{\}g sono quasi sempre la scelta migliore.

```
1 #include <stdio.h>
2
3 int main(void) {
4     double avogadro = 6.022e23;
5     double planck = 6.626e-34;
6
7     printf("Numero di Avogadro:\n");
8     printf("%f\n", avogadro);
9     printf("%e\n", avogadro);
10    printf("%g\n", avogadro);
11
12    printf("Costante di Planck:\n");
13    printf("%f\n", planck);
14    printf("%e\n", planck);
15    printf("%g\n", planck);
16
17    return 0;
18 }
```

Soluzione

LEZIONE 4

Strutture Decisionali e l'Iterazione Fondamentale



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 4

Strutture Decisionali e l'Iterazione Fondamentale

Nelle lezioni precedenti abbiamo completato la nostra cassetta degli attrezzi con i tipi di dato, la codifica dei reali e della tabella ASCII, e i dettagli sui modificatori di dimensione. Finora, però, i nostri programmi sono stati per lo più sequenze lineari di istruzioni. In questa lezione scopriremo come alterare questo flusso per prendere decisioni complesse e ripetere porzioni di codice. Partiremo analizzando come la CPU esegue i salti a basso livello (Assembly e `goto`), vedremo come il teorema di Böhm-Jacopini ci garantisca la possibilità di scrivere codice pulito senza usare salti incondizionati (spaghetti code), e impareremo a usare la selezione (`if/else`, `switch`, operatore ternario) e l'iterazione base (`while` con `break/continue`) in C. Infine, faremo la nostra prima conoscenza con il debugger **LLDB** per ispezionare il comportamento del codice in tempo reale.

4.1 Dalla RAM alla Programmazione Strutturata (Sotto il Cofano)

4.1.1 Il Flusso non Lineare e il Trucco del Salto (Goto)

Come fa il computer a saltare delle righe di codice o a ripetere un blocco di istruzioni? Fisicamente, la CPU possiede un registro speciale chiamato **Program Counter (PC)** che memorizza l'indirizzo della RAM dell'istruzione attualmente in esecuzione.

Normalmente il PC avanza in sequenza, istruzione dopo istruzione. Per deviare questo comportamento, la CPU non conosce blocchi o parentesi graffe: esegue semplicemente dei **salti (jump)** caricando un indirizzo diverso all'interno del Program Counter.

Consideriamo questa struttura condizionale in C e come viene convertita in linguaggio macchina dal compilatore:

```
1 /* Codice sorgente in C */
2 int x = 5;
3 if (x == 5) {
4     x = 10;
5 }
```

```
1 ; Equivalente Assembly concettuale (con il registro R1 che contiene il
   ↪ valore di x)
```

```

2  cmp R1, 5          ; Confronta il valore in R1 con il letterale 5
3  jne FINE_IF        ; Jump if Not Equal: se non sono uguali, salta a
   ↪ FINE_IF!
4  mov R1, 10         ; Se R1 == 5, esegui questo blocco (scrivi 10 in R1)
5  FINE_IF:
6  ; Il programma continua...

```

Nota

L'if tradotto al contrario Il compilatore traduce l'istruzione if al contrario: genera un'istruzione di salto condizionato (jne - salta se non sono uguali) che evita il blocco condizionale quando la condizione è **falsa**.

4.1.2 La Flessibilità del goto e lo Spaghetti Code

Anche in C è possibile scrivere salti incondizionati di basso livello abbinando l'istruzione goto a delle etichette:

```

1  int x = 10;
2  if (x > 5) goto SALTO;
3  printf("Questo messaggio viene ignorato.\n");
4
5  SALTO:
6  printf("Siamo saltati direttamente qui.\n");

```

Con il goto possiamo saltare in avanti (per saltare porzioni di codice) ma anche all'indietro (creando dei cicli). Negli anni '60, l'uso indiscriminato di questi salti liberi all'interno del codice portava a programmi disordinati definiti **spaghetti code**, complessi da comprendere e debuggare.

4.1.3 Il Teorema di Böhm-Jacopini

Nel 1966, gli informatici italiani **Corrado Böhm** e **Giuseppe Jacopini** pubblicarono una dimostrazione fondamentale che pose fine al caos dei salti non regolamentati:

Attenzione

Teorema di Böhm-Jacopini Qualsiasi algoritmo computabile può essere espresso utilizzando esclusivamente tre strutture di controllo fondamentali:

1. **Sequenza:** esecuzione ordinata di istruzioni una dopo l'altra.
2. **Selezione:** deviazione condizionale del flusso (es. bivio sì/no).
3. **Iterazione:** ripetizione di un blocco finché una condizione rimane vera.

Questo teorema assicura che possiamo bandire l'uso del goto senza perdere alcuna potenza computazionale, affidando la logica del nostro codice a strutture pulite e nidificate.

4.1.4 Concettualizzare con i Diagrammi di Flusso (Flowchart)

I diagrammi di flusso sono la visualizzazione grafica immediata delle tre strutture di controllo:

- **Rettangolo:** indica un'istruzione sequenziale.
- **Rombo:** indica una selezione condizionale che dirama il flusso.
- **Frecce all'indietro:** indicano un loop iterativo che ritorna in un punto precedente del programma.

```
1 graph TD
2   Start([Inizio]) --> RiempiCaldaia[Riempi la caldaia d'acqua]
3   RiempiCaldaia --> InserisciFiltro[Inserisci il filtro e il caffè?]
4   InserisciFiltro --> ChiudiMoka[Chiudi la moka e mettila sul fuoco]
5   ChiudiMoka --> CheckUscito{Il caffè ? uscito?}
6   CheckUscito -- No --> Attendi[Rimani in attesa]
7   Attendi --> CheckUscito
8   CheckUscito -- S? --> SpegniFuoco[Spegni il fuoco]
9   SpegniFuoco --> Versa[Versa il caffè? nella tazzina]
10  Versa --> End([Fine])
```

Il diagramma mostra chiaramente le tre strutture all'opera:

1. **Sequenza:** I passaggi iniziali si susseguono linearmente (Riempi caldaia → Inserisci filtro → Chiudi moka).
2. **Selezione:** Il rombo decisore (Il caffè è uscito?) separa il flusso in due percorsi alternativi basati su una risposta logica (Sì/No).
3. **Iterazione:** Il percorso che dal blocco di attesa ritorna al controllo del rombo forma un anello chiuso (ciclo) che ripete l'azione finché la condizione non cambia.

Suggerimento

Non esagerare con i disegni Non useremo software complessi per disegnare diagrammi di flusso: sono strumenti utili dal punto di vista concettuale ma non si usano nella progettazione di software complessi. Ci basta saperli tracciare rapidamente a mano libera.

4.2 Le Strutture Condizionali in C

4.2.1 La Selezione Completa: `if`, `else` ed `else if`

Completiamo la sintassi del blocco condizionale introducendo la via alternativa (`else`) e la cascata di controlli escludentisi (`else if`):


```

1  if (punti >= 90) {
2      printf("Eccellente\n");
3  } else if (punti >= 60) {
4      printf("Sufficiente\n");
5  } else {
6      printf("Non sufficiente\n");
7  }

```

Quando una delle condizioni a cascata risulta vera, la CPU esegue il relativo blocco e salta immediatamente oltre la fine della struttura dell'intera catena di `if`, ignorando i test successivi.

Nota

Graffe e istruzioni singole Se il blocco di codice contiene **una sola istruzione**, le parentesi graffe `{ } { }` sono opzionali:

```

1  if (temperatura > 30)
2      printf("Fa caldo\n");
3  else
4      printf("Si sta bene\n");

```

Tuttavia, è considerata una buona norma di programmazione inserire sempre le graffe per evitare errori logici di associazione quando si aggiungono istruzioni successivamente.

4.2.2 Selezione Multipla Discreta: `switch`

Quando dobbiamo confrontare una singola variabile discreta (es. intero o carattere) con molti valori fissi possibili, l'uso di molti `else if` risulta ridondante. Il C offre il costrutto `switch`:

```

1  #include <stdio.h>
2
3  int main(void) {
4      int scelta;
5      printf("---- MENU DISTRIBUTORE ---\n");
6      printf("1. Caff? (0.50?)\n");
7      printf("2. T? caldo (0.60?)\n");
8      printf("3. Cioccolata (0.70?)\n");
9      printf("Inserisci la tua scelta (1-3): ");
10     scanf("%d", &scelta);
11
12     switch (scelta) {
13         case 1:
14             printf("Hai selezionato: Caff?\n");
15             break;
16         case 2:
17             printf("Hai selezionato: T? caldo\n");
18             break;
19         case 3:
20             printf("Hai selezionato: Cioccolata\n");
21             break;
22         default:

```

```

23         printf("Selezione errata!\n");
24     }
25     return 0;
26 }

```

Attenzione

Il trabocchetto del Fall-Through Se omettete l'istruzione `break` alla fine di un `case`, la CPU non si ferma! Continuerà a eseguire a cascata le istruzioni dei `case` successivi anche se le loro condizioni non sono verificate. Questo comportamento può essere sfruttato intenzionalmente (ad esempio per far eseguire la stessa azione a 'a' e 'A'), ma nella maggior parte dei casi l'omissione del `break` è un bug.

Vincoli dello switch

- Può valutare **solo tipi discreti** (`int`, `char`, `short`, `long`). Non si possono usare `switch` con stringhe o con numeri a virgola mobile (`double`).
- Può verificare unicamente condizioni di **uguaglianza**, non relazioni d'ordine (non potete scrivere `case >= 18:`).

Nota

Lo `switch` non è indispensabile. Qualsiasi `switch` può essere riscritto in modo equivalente usando `if` ed `else if` annidati. È semplicemente uno strumento comodo per migliorare la leggibilità del codice.

4.2.3 L'Operatore Ternario (?:)

Consente di scrivere una selezione condizionale rapida per assegnare un valore a una variabile:

```

1  /* Con operatore ternario */
2  int max = (a > b) ? a : b;
3
4  /* Con if-else equivalente */
5  int max;
6  if (a > b) {
7      max = a;
8  } else {
9      max = b;
10 }

```

La sintassi è: `(condizione) ? valore_se_vero : valore_se_falso;`

4.3 L'iterazione Fondamentale: Il Ciclo while

4.3.1 Il Costrutto `while`

Il ciclo `while` ripete l'esecuzione del blocco di codice racchiuso tra le graffe finché la sua condizione di controllo rimane vera (diversa da zero).

```
1 while (condizione) {
2     // Codice da ripetere
3 }
```

Attenzione

L'universalità del `while` Nelle prossime lezioni studieremo altri tipi di cicli come `for` o `do-while`. È fondamentale comprendere che **tutto ciò che si può implementare con gli altri cicli può essere espresso con un ciclo `while`**. Esso rappresenta l'iterazione base universale.

4.3.2 Evitare il Loop Infinito

Affinché il ciclo termini, all'interno del blocco delle graffe deve esistere un'istruzione che modifichi lo stato delle variabili coinvolte nella condizione di controllo, facendola diventare prima o poi falsa (0). Altrimenti il programma rimarrà bloccato in un loop infinito.

Vediamo un esempio di divisione intera eseguita per sottrazioni successive senza usare l'operatore `/`:

```
1 #include <stdio.h>
2
3 int main(void) {
4     int dividendo = 17;
5     int divisore = 5;
6
7     int quoziente = 0;
8     int resto = dividendo;
9
10    // Sottraiamo finché il resto è maggiore o uguale al divisore
11    while (resto >= divisore) {
12        resto = resto - divisore; // Aggiorna lo stato!
13        quoziente++;
14    }
15
16    printf("Quoziente: %d, Resto: %d\n", quoziente, resto);
17    return 0;
18 }
```

4.3.3 Algoritmo del Massimo e Minimo (Sequenza Fissa)

Prima di addentrarci in cicli dal comportamento dinamico, studiamo un problema classico e fondamentale dell'informatica: **trovare il valore massimo e minimo all'interno di una sequenza di numeri** inseriti dall'utente.

Immaginiamo di dover leggere esattamente 10 numeri interi positivi. Per trovare gli estremi (massimo e minimo) senza dover memorizzare tutti i numeri contemporaneamente, applichiamo un algoritmo iterativo basato su **valori provvisori**:

1. Acquisiamo il primo numero e lo impostiamo provvisoriamente sia come massimo che come minimo.
2. Per ciascuno dei successivi 9 numeri: lo confrontiamo con il massimo provvisorio (se è maggiore, aggiorniamo il massimo) e con il minimo provvisorio (se è minore, aggiorniamo il minimo).

```
1  #include <stdio.h>
2
3  int main(void) {
4      int numero;
5      int max;
6      int min;
7
8      // Leggiamo il primo numero per impostare lo stato iniziale
9      printf("Inserisci il 1? numero: ");
10     if (scanf("%d", &numero) != 1) {
11         printf("Errore di lettura.\n");
12         return 1;
13     }
14     max = numero;
15     min = numero;
16
17     // Ciclo per i restanti 9 numeri
18     int i = 2; // Contatore: indica quale numero stiamo leggendo
19     while (i <= 10) {
20         printf("Inserisci il %d? numero: ", i);
21         if (scanf("%d", &numero) != 1) {
22             printf("Errore di lettura.\n");
23             return 1;
24         }
25
26         if (numero > max) {
27             max = numero; // Aggiorna il massimo provvisorio
28         }
29         if (numero < min) {
30             min = numero; // Aggiorna il minimo provvisorio
31         }
32         i++; // Passa al numero successivo
33     }
34
35     printf("\n--- Risultati ---\n");
36     printf("Il numero massimo inserito ?: %d\n", max);
37     printf("Il numero minimo inserito ?: %d\n", min);
38
39     return 0;
40 }
```

Suggerimento

Perché inizializzare con il primo elemento? Un errore comune nei principianti è inizializzare il massimo a 0 e il minimo a valori arbitrari (es. 99999). Questa scelta è rischiosa: se l'utente inserisse solo numeri negativi, il massimo rimarrebbe erroneamente 0 (che non fa parte della sequenza). Inizializzare `max` e `min` con il primo valore inserito garantisce la correttezza dell'algoritmo indipendentemente dal range dei dati.

4.3.4 Modificare il Flusso del Ciclo: `break` e `continue`

Possiamo alterare il comportamento standard di un ciclo a metà dell'esecuzione utilizzando due istruzioni speciali:

- **`break`**: esce immediatamente dal ciclo `while`, saltando alla prima istruzione utile dopo il blocco.
- **`continue`**: interrompe l'iterazione corrente a metà, saltando le righe sottostanti per tornare subito in cima a valutare nuovamente la condizione di controllo.

Per comprendere l'utilità di questi costrutti, analizziamo un problema reale: scrivere un programma che legga una sequenza di voti d'esame universitari inseriti da tastiera. Il programma deve:

1. Accettare solo voti validi compresi tra 18 e 30 (incluso 31 per indicare 30 e lode), saltando qualsiasi inserimento errato tramite l'istruzione **`continue`**.
2. Interrompere l'acquisizione non appena l'utente digita il valore di sentinella -1 tramite l'istruzione **`break`**.
3. Calcolare e visualizzare alla fine: il numero di esami validi inseriti, il voto massimo, il voto minimo e la media aritmetica dei voti validi.

```

1  #include <stdio.h>
2
3  int main(void) {
4      int voto = 0;
5      int somma = 0;
6      int conteggio = 0;
7
8      // Inizializziamo il massimo con il valore pi? basso possibile per un
      ↪ esame
9      int max = 0;
10     // Inizializziamo il minimo con il valore pi? alto possibile per un
      ↪ esame
11     int min = 100;
12
13     printf("Inserisci i voti degli esami (compresi tra 18 e 31; inserisci
      ↪ -1 per terminare):\n");
14
15     while (1) { // Ciclo potenzialmente infinito, controllato all'interno

```

```

16     printf("Inserisci voto: ");
17     if (scanf("%d", &voto) != 1) {
18         printf("Inserimento non valido.\n");
19         break;
20     }
21
22     // Caso di uscita volontaria
23     if (voto == -1) {
24         break; // Esce immediatamente dal ciclo
25     }
26
27     // Filtro di validit?: i voti validi vanno da 18 a 30 (31 per
↪ lode)
28     if (voto < 18 || voto > 31) {
29         printf("Voto non valido! Deve essere compreso tra 18 e 31 (31
↪ = 30L).\n");
30         continue; // Salta il resto del corpo del ciclo e torna in
↪ cima
31     }
32
33     // Se siamo arrivati qui, il voto ? valido. Aggiorniamo le
↪ statistiche:
34     somma = somma + voto;
35     conteggio++;
36
37     if (voto > max) {
38         max = voto;
39     }
40     if (voto < min) {
41         min = voto;
42     }
43 }
44
45 // Calcolo e stampa dei risultati
46 if (conteggio > 0) {
47     double media = (double)somma / conteggio;
48     printf("\n--- Statistiche Esami ---\n");
49     printf("Esami superati: %d\n", conteggio);
50     printf("Voto massimo: %d\n", max);
51     printf("Voto minimo: %d\n", min);
52     printf("Media voti: %.2f\n", media);
53 } else {
54     printf("\nNessun voto valido inserito.\n");
55 }
56
57 return 0;
58 }

```

Nota

Perché qui `max = 0` e `min = 100` vanno bene? Poco sopra abbiamo sconsigliato di inizializzare massimo e minimo con valori arbitrari. Qui però l'intervallo dei dati è **noto e garantito dal filtro**: ogni voto che arriva all'aggiornamento di `max` e `min` è compreso

tra 18 e 31. Quindi il primo voto valido sarà sicuramente maggiore di 0 e minore di 100, e sostituirà entrambi i valori iniziali. Inoltre, se non viene inserito alcun voto valido, max e min non vengono mai stampati grazie al controllo `conteggio > 0`.

4.4 Quiz di Autovalutazione

Quiz 1

Domanda

1. Quale registro della CPU memorizza fisicamente l'indirizzo dell'istruzione in esecuzione?

- A) Il registro Accumulatore.
- B) Il Program Counter (PC).
- C) Il registro Stack Pointer.
- D) Il registro di Stato (Flags).

Soluzione
Risposta corretta: B
Spiegazione: Il Program Counter memorizza l'indirizzo di memoria dell'istruzione corrente e viene incrementato per puntare alla successiva o modificato direttamente durante le istruzioni di salto.

Quiz 2

Domanda

2. Cosa stabilisce il Teorema di Böhm-Jacopini?

- A) Che tutti i calcolatori devono seguire l'architettura di Von Neumann.
- B) Che qualsiasi algoritmo può essere espresso usando solo Sequenza, Selezione e Iterazione.
- C) Che il compilatore C converte i tipi reali sempre usando il bias 1023.
- D) Che l'uso del `goto` è obbligatorio per scrivere cicli infiniti.

Soluzione
Risposta corretta: B
Spiegazione: Dimostrato nel 1966, garantisce che i tre costrutti base siano sufficienti per esprimere qualsiasi logica computazionale, rendendo l'istruzione di salto incondizionato `goto` non necessaria.

Quiz 3

Domanda

3. Cosa succede se si omette il comando `break` in uno dei blocchi `case` di uno `switch`?

- A) Il compilatore segnala un errore di sintassi e arresta la compilazione.
- B) Il programma crasha con un errore di segmentation fault.
- C) La CPU prosegue eseguendo le istruzioni dei `case` successivi a cascata (fall-through).
- D) Lo `switch` viene convertito automaticamente in una divisione intera.

Soluzione
Risposta corretta: C
Spiegazione: Il `break` serve a uscire dallo `switch`. Se manca, l'esecuzione prosegue linearmente sulle righe dei casi successivi, indipendentemente dal valore dei loro controlli.

Quiz 4

Domanda

4. Quale dei seguenti tipi non può essere valutato all'interno di una selezione `switch`?

- A) `int`
- B) `char`
- C) `double`
- D) `short`

Soluzione
Risposta corretta: C
Spiegazione: Il costrutto `switch` in C opera esclusivamente su valori discreti convertibili in interi. I numeri in virgola mobile (`double` e `float`) e le stringhe sono esclusi.

Quiz 5

Domanda

5. Quale espressione con l'operatore ternario equivale a questo blocco?

```
1 int ris;  
2 if (valore > 0) ris = 1;  
3 else ris = 0;
```

- A) `int ris = (valore > 0) ? 0 : 1;`

- B) `int ris = (valore > 0) ? 1 : 0;`
- C) `int ris = (valore > 0) : 1 ? 0;`
- D) `int ris = valore > 0;`

Soluzione
Risposta corretta: B (anche la D è corretta in C, ma la B usa esplicitamente l'operatore ternario)
Spiegazione: La sintassi dell'operatore ternario assegna il primo valore (1) se la condizione è vera, altrimenti il secondo (0).

Quiz 6

Domanda

6. Che cos'è l'universalità del ciclo `while`?

- A) Significa che funziona su qualsiasi sistema operativo.
- B) Significa che qualsiasi ciclo scritto con `for` o `do-while` può essere espresso tramite il `while`.
- C) Significa che il `while` non può mai produrre un ciclo infinito.
- D) Significa che può valutare solo condizioni logiche ma non confronti numerici.

Soluzione
Risposta corretta: B
Spiegazione: Il `while` rappresenta la forma più pura ed elementare di iterazione. Gli altri cicli sono zuccheri sintattici che facilitano la scrittura, ma non aggiungono potenza di calcolo.

Quiz 7

Domanda

7. Cosa fa l'istruzione `continue` all'interno di un ciclo `while`?

- A) Esce immediatamente dal ciclo saltando alla fine del programma.
- B) Salta la parte rimanente del blocco del ciclo solo per l'iterazione corrente, ritornando in cima a valutare la condizione.
- C) Esegue di nuovo l'istruzione precedente per correggere eventuali errori.
- D) Azzera il contatore del ciclo portando la condizione a vero.

Soluzione
Risposta corretta: B
Spiegazione: `continue` interrompe solo il passaggio corrente. Riporta il flusso in cima al controllo del ciclo per l'iterazione successiva.

Quiz 8

Domanda

8. Nel programma di esempio sui voti universitari, cosa succede se l'utente inserisce la sequenza di numeri: 25, 12, 30, -1?

- A) Il programma stampa una media pari a 22.33 (escludendo solo -1).
- B) Il programma termina restituendo un errore a causa del valore 12.
- C) Il programma ignora il voto 12 (tramite `continue`) e calcola le statistiche (media 27.50) su 25 e 30 dopo l'uscita tramite `break` con -1.
- D) Il programma va in loop infinito.

Soluzione
Risposta corretta: C
Spiegazione: Il valore 12 è fuori dall'intervallo [18, 31], quindi attiva la condizione d'errore e l'istruzione `continue` che salta le istruzioni di accumulo e conteggio. Il valore -1 provoca l'attivazione del `break` che interrompe il ciclo e avvia la stampa finale delle medie basate solo sui voti validi (25 e 30, la cui media è 27.50).

4.5 Esercizi Pratici

Esercizio 1: Il Calcolo del Massimo tra tre numeri

Domanda

Scrivete un programma che legga tre numeri interi da tastiera e stampi il valore massimo tra i tre, usando una cascata di `if/else` ed evitando l'uso di operatori logici complessi.

Spiegazione dell'algoritmo: Assumiamo inizialmente che a sia il massimo. Successivamente confrontiamo questo valore provvisorio con b ed eventualmente lo aggiorniamo, ripetendo lo stesso passaggio con c.

```

1  #include <stdio.h>
2
3  int main(void) {
4      int a, b, c;
5      printf("Inserisci tre numeri interi: ");
6      scanf("%d %d %d", &a, &b, &c);
7
8      int max = a;
9
10     if (b > max) {
11         max = b;
12     }
13     if (c > max) {
14         max = c;
15     }
16     printf("Il valore massimo inserito è: %d\n", max);
17     return 0;
18 }
19

```

Soluzione

Esercizio 2: Calcolatore di Tariffe Distributore

Domanda

Scrivete un programma che simuli la selezione di una scelta menu (1=Caffè, 2=Cioccolata, 3=Tè) e stampi il costo, usando lo `switch` con gestione del caso di errore tramite `default`.

Spiegazione dell'algoritmo: Il costrutto switch seleziona il blocco corretto controllando l'intero scelta. L'istruzione break garantisce l'uscita dallo switch evitando l'esecuzione del codice sottostante.

```

1  #include <stdio.h>
2
3  int main(void) {
4      int scelta;
5      printf("Menu Bevande:\n1. Caffè\n2. Cioccolata\n3. Tè\nscelta:");
6      scanf("%d", &scelta);
7
8      switch (scelta) {
9          case 1:
10             printf("Costo Caffè: 0.50?\n");
11             break;
12          case 2:
13             printf("Costo Cioccolata: 0.70?\n");
14             break;
15          case 3:
16             printf("Costo Tè: 0.60?\n");
17             break;
18          default:
19             printf("Errore: Selezione non valida!\n");
20             }
21     return 0;
22 }

```

Soluzione

Esercizio 3: Divisione intera senza l'operatore di

Domanda

divisione

Scrivete un programma che calcoli la divisione intera e il resto tra due numeri interi positivi usando solo sottrazioni ripetute all'interno di un ciclo while.

Spiegazione dell'algoritmo: Sottraiamo il divisore dal dividendo cumulato in resto fino a quando non è più possibile sottrarre senza andare in negativo. Il numero di sottrazioni effettuate corrisponde al quoziente.

```

1  #include <stdio.h>
2
3  int main(void) {
4      int dividendo, divisore;
5      printf("Inserisci dividendo e divisore positivi: ");
6      scanf("%d %d", &dividendo, &divisore);
7
8      if (divisore <= 0 || dividendo < 0) {
9          printf("Errore: inserisci valori validi!\n");
10         return 1;
11     }
12
13     int quoziente = 0;
14     int resto = dividendo;
15
16     while (resto >= divisore) {
17         resto = resto - divisore;
18         quoziente++;
19     }
20
21     printf("Risultato: %d con resto %d\n", quoziente, resto);
22     return 0;
23 }
```

Soluzione

Esercizio 4: Massimo Comun Divisore (MCD)

Domanda

Implementate l'algoritmo di Euclide a sottrazioni successive per calcolare il MCD tra due numeri interi positivi letti da tastiera.

Spiegazione dell'algoritmo: Ripetiamo la sottrazione del minore dal maggiore finché le due variabili temp\{}_a e temp\{}_b non contengono lo stesso valore, che per definizione di Euclide corrisponde al MCD.

```

1  #include <stdio.h>
2
3  int main(void) {
4      int a, b;
5      printf("Inserisci due numeri interi positivi: ");
6      scanf("%d", &a, &b);
7
8      if (a <= 0 || b <= 0) {
9          printf("I numeri devono essere positivi!\n");
10         return 1;
11     }
12
13     int temp_a = a;
14     int temp_b = b;
15
16     while (temp_a != temp_b) {
17         if (temp_a > temp_b) {
18             temp_a = temp_a - temp_b;
19         } else {
20             temp_b = temp_b - temp_a;
21         }
22     }
23
24     printf("Il MCD tra %d e %d è %d\n", a, b, temp_a);
25     return 0;
26 }

```

Soluzione

Esercizio 5: Stampa dei soli numeri dispari con continue

Domanda

Scrivete un programma che stampi tutti i numeri da 1 a 15, escludendo i numeri pari tramite l'uso dell'istruzione `continue` all'interno di un ciclo `while`.

Spiegazione dell'algoritmo: A ogni iterazione incrementiamo `i`. Se `i` è pari (`i % 2 == 0`), l'istruzione `continue` fa ripartire immediatamente il ciclo dal controllo, ignorando l'istruzione `printf` sottostante.

```

1  #include <stdio.h>
2
3  int main(void) {
4      int i = 0;
5      while (i < 15) {
6          i++;
7          if (i % 2 == 0) {
8              continue; // Salta il resto del ciclo e torna in cima
9          }
10         printf("%d ", i);
11     }
12     printf("\n");
13     return 0;
14 }

```

Soluzione

Esercizio 6: Somma di numeri fino a inserimento dello zero

Domanda

Scrivete un programma che continui a richiedere numeri interi all'utente sommandoli tra loro. Il ciclo deve interrompersi non appena l'utente digita il valore 0. Stampate infine la somma totale.

Spiegazione dell'algoritmo: Fin tanto che l'utente non inserisce 0, la condizione `numero != 0` rimane vera e il ciclo accumula il valore letto nella variabile `somma`.

```

1  #include <stdio.h>
2
3  int main(void) {
4      int numero = -1; // Inizializzato a un valore diverso da zero
5      int somma = 0;
6
7      while (numero != 0) {
8          printf("Inserisci un numero (0 per terminare): ");
9          scanf("%d", &numero);
10         somma = somma + numero;
11     }
12     printf("La somma totale dei numeri inseriti è: %d\n", somma);
13     return 0;
14 }
15

```

Soluzione

Esercizio 7: Conversione carattere maiuscolo/minuscolo

Domanda

Scrivete un programma che acquisisca un singolo carattere da tastiera. Se è una lettera maiuscola, stampate la rispettiva lettera minuscola sfruttando la tabella ASCII (senza usare funzioni di libreria), altrimenti stampate un errore.

Spiegazione dell'algoritmo: Sfruttiamo la codifica ordinata ASCII. Le lettere maiuscole e minuscole distano sempre 32 posizioni tra loro. Sommando questa differenza passiamo al carattere minuscolo equivalente.

```

1  #include <stdio.h>
2
3  int main(void) {
4      char c;
5      printf("Inserisci un carattere maiuscolo: ");
6      scanf("%c", &c); // Lo spazio prima di %c consuma eventuali
                          // whitespace residui
7
8      if (c >= 'A' && c <= 'Z') {
9          // La distanza tra maiuscole e minuscole in ASCII è
          // costante ('a' - 'A' = 32)
10         char minuscolo = c + ('a' - 'A');
11         printf("Lettera minuscola: %c\n", minuscolo);
12     } else {
13         printf("Errore: il carattere inserito non è una lettera
14             maiuscola!\n");
15     }
16     return 0;
17 }
```

Soluzione

Esercizio 8: Menu Distributore Ciclico con Uscita

Domanda

volontaria

Scrivete un programma che simuli il distributore di bevande dell'Esercizio 2 all'interno di un ciclo `while`. Il menu deve continuare a ripresentarsi all'utente dopo ogni acquisto finché non viene selezionata l'opzione di uscita (tasto 0).

Soluzione

```

1  #include <stdio.h>
2
3  int main(void) {
4      int scelta = -1;
5
6      while (scelta != 0) {
7          printf("\n--- DISTRIBUTORE AUTOMATICO ---\n");
8          printf("1. Caffè (0.50?)\n");
9          printf("2. Cioccolata (0.70?)\n");
10         printf("3. Tè (0.60?)\n");
11         printf("0. Esci\n");
12         printf("Scelta: ");
13         scanf("%d", &scelta);
14
15         switch (scelta) {
16             case 0:
17                 printf("Arrivederci!\n");
18                 break;
19             case 1:
20                 printf("Prezzo: 0.50?\n");
21                 break;
22             case 2:
23                 printf("Prezzo: 0.70?\n");
24                 break;
25             case 3:
26                 printf("Prezzo: 0.60?\n");
27                 break;
28             default:
29                 printf("Scelta errata!\n");
30         }
31     }
32     return 0;
33 }

```

Spiegazione dell'algoritmo: Il ciclo `while` continua a riproporre il menu fino a quando `scelta != 0` diventa falsa. Lo `switch` all'interno gestisce le opzioni di acquisto e l'opzione di uscita.

Esercizio 9: Calcolo potenza senza pow**Domanda**

Scrivete un programma che calcoli la potenza base elevato esponente (b^e) per esponenti interi non negativi, usando un ciclo `while` e moltiplicazioni ripetute.

Soluzione

```

1  #include <stdio.h>
2
3  int main(void) {
4      double base;
5      int esponente;
6      printf("Inserisci la base (reale) e l'esponente (intero)
    ↳ positivo: ");
7      scanf("%lf %d", &base, &esponente);
8
9      if (esponente < 0) {
10         printf("L'esponente deve essere non negativo!\n");
11         return 1;
12     }
13
14     double risultato = 1.0;
15     int i = 0;
16
17     while (i < esponente) {
18         risultato = risultato * base;
19         i++;
20     }
21
22     printf("Risultato: %.6f\n", risultato);
23     return 0;
24 }

```

Spiegazione dell'algoritmo: Inizializziamo il risultato a 1.0. Moltiplichiamo il risultato per la base tante volte quante indichiate dall'esponente, usando la variabile contatore `i` come controllo.

Esercizio 10: Fibonacci Limitato

Domanda

Scrivete un programma che stampi tutti i numeri della celebre successione di Fibonacci (1, 1, 2, 3, 5, 8, 13...) che siano inferiori o uguali a un valore massimo inserito dall'utente, usando un ciclo `while`.

Soluzione

```

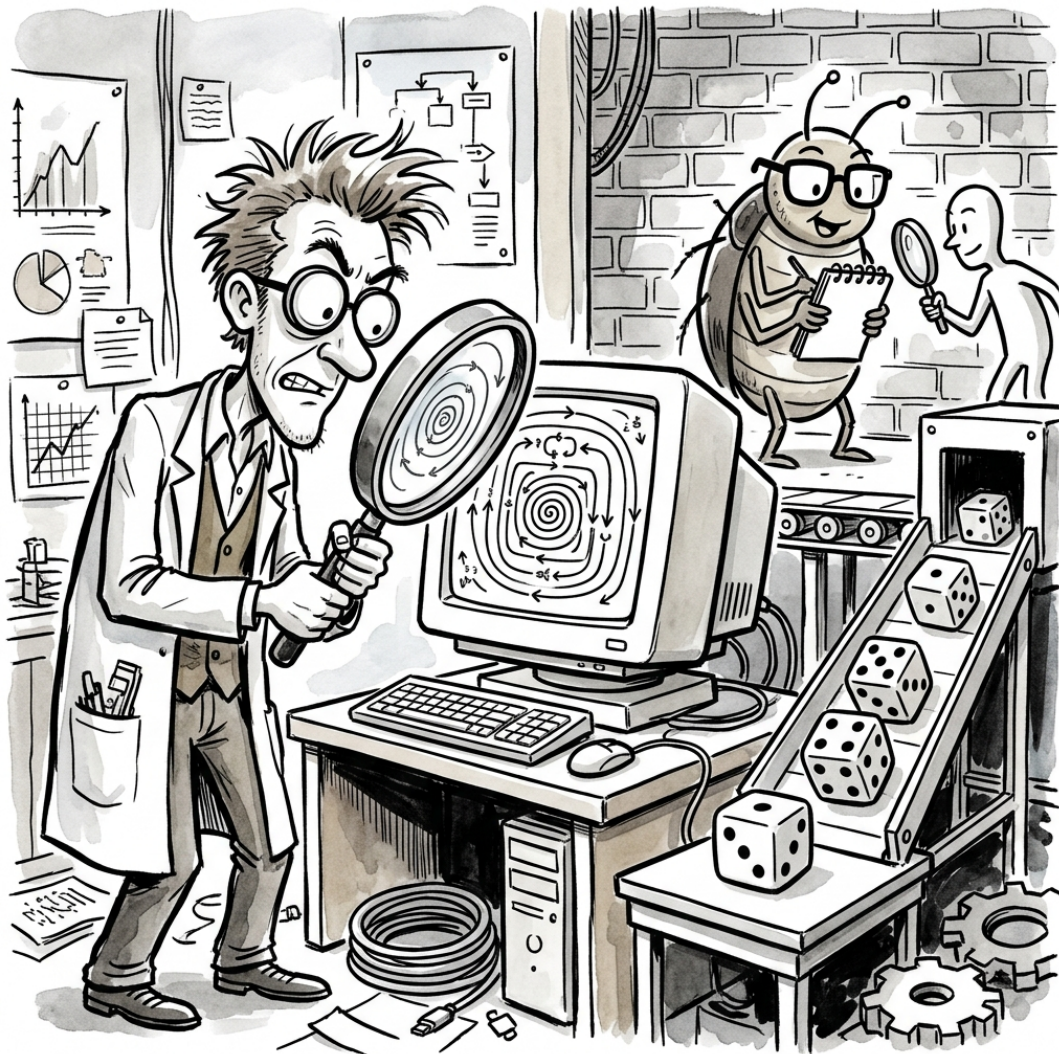
1  #include <stdio.h>
2
3  int main(void) {
4      int limite;
5      printf("Inserisci il valore massimo: ");
6      scanf("%d", &limite);
7
8      int a = 1;
9      int b = 1;
10
11     printf("Successione di Fibonacci: ");
12     while (a <= limite) {
13         printf("%d ", a);
14         int successivo = a + b;
15         a = b;
16         b = successivo;
17     }
18     printf("\n");
19     return 0;
20 }

```

Spiegazione dell'algoritmo: La successione di Fibonacci si ottiene sommando i due termini precedenti. A ogni passaggio calcoliamo il successivo, facciamo slittare in avanti i valori delle variabili *a* e *b* e ci fermiamo non appena superiamo la soglia impostata.

LEZIONE 5

Iterazione Avanzata, Visibilità, Casualità e Debugging



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 5

Iterazione Avanzata, Visibilità, Casualità e Debugging

Nelle lezioni precedenti abbiamo appreso il funzionamento delle strutture condizionali e del ciclo `while` come pilastri del controllo del flusso. In questa lezione completeremo lo studio dell'iterazione introducendo il costrutto `for` (ideale per cicli controllati da contatore) e il costrutto `do-while` (per cicli post-condizionati che devono essere eseguiti almeno una volta). Approfondiremo come la visibilità e la durata delle variabili siano determinate dal blocco di codice in cui vengono dichiarate (scope). Vedremo poi come simulare la casualità nei nostri programmi tramite generatori pseudo-casuali (`rand` e `srand`). Infine, affronteremo uno degli strumenti più importanti per lo sviluppo software: l'uso del debugger **LLDB** sia da terminale sia graficamente in VS Code.

5.1 Iterazione Avanzata: I Cicli `for` e `do-while`

5.1.1 Il Ciclo `for` (Iterazione controllata da contatore)

Il ciclo `for` è un'abbreviazione sintattica ("zucchero sintattico") del ciclo `while`. Viene impiegato quando il numero di ripetizioni è noto a priori o quando il ciclo è guidato da una variabile che funge da contatore.

La sintassi generale è:

```
1 for (inizializzazione; condizione; incremento) {  
2     // Corpo del ciclo  
3 }
```

Questo blocco equivale esattamente alla seguente struttura realizzata con un ciclo `while`:

```
1 inizializzazione;  
2 while (condizione) {  
3     // Corpo del ciclo  
4     incremento;  
5 }
```

Esempio: Stampa dei numeri pari da 2 a N

```
1  #include <stdio.h>
2
3  int main(void) {
4      int limit = 10;
5
6      // Inizializzazione di i, controllo della condizione, e incremento
        ↪ eseguiti in un'unica riga
7      for (int i = 2; i <= limit; i += 2) {
8          printf("%d ", i);
9      }
10     printf("\n");
11     return 0;
12 }
```

La variabile `i` dichiarata all'interno dell'inizializzazione del `for` è locale al ciclo stesso. Come vedremo più avanti, non è possibile fare riferimento ad essa al di fuori del corpo del `for`.

Il `while` equivalente

Per comprendere meglio la mappatura del `for` sul `while`, ecco come si scrive lo stesso programma usando l'iterazione base:

```
1  #include <stdio.h>
2
3  int main(void) {
4      int limit = 10;
5
6      int i = 2;           // 1. Inizializzazione
7      while (i <= limit) { // 2. Condizione
8          printf("%d ", i);
9          i += 2;          // 3. Incremento
10     }
11     printf("\n");
12     return 0;
13 }
```

5.1.2 Il Ciclo `do-while` (Iterazione post-condizionata)

A differenza dei cicli `while` e `for` che valutano la condizione *prima* di eseguire il corpo del ciclo (e quindi potrebbero non essere eseguiti affatto se la condizione iniziale è falsa), il ciclo `do-while` valuta la condizione **dopo** l'esecuzione del corpo.

Questo garantisce che le istruzioni all'interno del ciclo vengano eseguite **almeno una volta**.

La sintassi è:

```
1  do {
2      // Corpo del ciclo (eseguito almeno una volta)
3  } while (condizione);
```

(Nota il punto e virgola `;` obbligatorio alla fine della tonda).

Confronto didattico: Validazione dell'input

La validazione dell'input da tastiera è lo scenario ideale per il ciclo do-while. Vediamo il confronto tra le due implementazioni:

Opzione A: Con il ciclo while (ridondante o con valori fittizi)

```

1  #include <stdio.h>
2
3  int main(void) {
4      int voto = 0; // Inizializzazione fittizia per forzare l'ingresso nel
    ↪ loop
5
6      printf("Inserisci il voto dell'esame (18-30): ");
7      scanf("%d", &voto);
8
9      while (voto < 18 || voto > 30) {
10         printf("Voto non valido! Inserisci il voto dell'esame (18-30): ")
    ↪ ;
11         scanf("%d", &voto); // Duplicazione del codice di lettura
12     }
13
14     printf("Voto registrato: %d\n", voto);
15     return 0;
16 }
```

Opzione B: Con il ciclo do-while (lineare e pulito)

```

1  #include <stdio.h>
2
3  int main(void) {
4      int voto; // Nessuna necessità di inizializzazione fittizia
5
6      do {
7          printf("Inserisci il voto dell'esame (18-30): ");
8          scanf("%d", &voto);
9
10         // Se il voto inserito non è valido, stampiamo un avviso prima di
    ↪ ripetere il ciclo
11         if (voto < 18 || voto > 30) {
12             printf("Voto non valido! Riprova.\n\n");
13         }
14     } while (voto < 18 || voto > 30);
15
16     printf("Voto registrato con successo: %d\n", voto);
17     return 0;
18 }
```

In questo modo otteniamo il meglio dei due mondi: un codice elegante privo di duplicazioni I/O, che però notifica l'errore all'utente in modo chiaro.

5.2 Cicli Annidati e Visibilità delle Variabili (Scope & Lifetime)

5.2.1 Cicli Annidati (Nested Loops)

È possibile inserire un ciclo all'interno del corpo di un altro ciclo. Questa struttura è fondamentale per elaborare griglie bidimensionali, matrici o combinazioni.

Esempio: La Griglia di Coordinate

```
1  #include <stdio.h>
2
3  int main(void) {
4      int righe = 3;
5      int colonne = 4;
6
7      for (int r = 1; r <= righe; r++) {          // Ciclo esterno
8          for (int c = 1; c <= colonne; c++) {    // Ciclo interno
9              printf("(%d,%d) ", r, c);
10             }
11             printf("\n"); // Andata a capo al termine di ogni riga
12         }
13         return 0;
14     }
```

Output:

```
1  (1,1) (1,2) (1,3) (1,4)
2  (2,1) (2,2) (2,3) (2,4)
3  (3,1) (3,2) (3,3) (3,4)
```

Il ciclo interno viene completato interamente per ciascuna iterazione del ciclo esterno.

5.2.2 Scope e Lifetime delle Variabili

Ogni variabile in C possiede uno **Scope** (l'ambito in cui il nome della variabile è visibile e utilizzabile) e un **Lifetime** (il tempo durante il quale la variabile occupa fisicamente uno spazio in memoria).

La regola del Block Scope (Scope di Blocco)

In C, ogni coppia di parentesi graffe `{ ... }` definisce un blocco di codice con un proprio scope locale:

- Qualsiasi variabile dichiarata all'interno di un blocco `{ ... }` esiste ed è visibile solo all'interno del blocco stesso e dei blocchi in esso nidificati.
- All'uscita del blocco, la variabile cessa di esistere e la sua memoria viene rilasciata.

Esempi di codice che non compilano

Caso 1: Variabile locale a una selezione condizionale

```

1  #include <stdio.h>
2
3  int main(void) {
4      int x = 10;
5      if (x > 5) {
6          int y = 20; // y ha scope limitato al blocco dell'if
7          printf("y: %d\n", y); // OK
8      }
9      // printf("y: %d\n", y); // ERRORE DI COMPILAZIONE! 'y' non esiste in
   ↪ questo scope.
10     return 0;
11 }
```

Caso 2: Variabile dichiarata dentro un ciclo

```

1  #include <stdio.h>
2
3  int main(void) {
4      for (int i = 0; i < 5; i++) {
5          int quadrato = i * i; // nasce e muore ad ogni singola iterazione
6      }
7      // printf("%d\n", quadrato); // ERRORE: 'quadrato' non esiste pi?!
8      // printf("%d\n", i);         // ERRORE: 'i' (contatore del for) non
   ↪ esiste pi?!
9      return 0;
10 }
```

Cenni sullo Stack di Memoria

Quando una funzione viene avviata (incluso il `main`), il sistema operativo riserva una porzione di memoria temporanea e organizzata chiamata **Stack** (pila). Le variabili locali dichiarate all'interno delle funzioni e dei blocchi vengono create inserendole in cima allo Stack. Non appena l'esecuzione esce dal blocco di appartenenza, queste variabili vengono eliminate liberando automaticamente la memoria dello Stack.

Anticipazione: Variabili Globali

Le variabili dichiarate all'esterno di qualsiasi funzione (fuori dal `main`) sono dette **variabili globali**. Esse vivono per tutta la durata del programma in una zona di memoria statica e possono essere lette e modificate da qualsiasi parte del codice. L'uso di variabili globali è considerato una pessima pratica perché rende il codice instabile, difficile da testare e pronò a bug logici. Nei laboratori e nelle prove d'esame il loro utilizzo è vietato.

5.3 Numeri Pseudo-Casuali e Debugging

5.3.1 Generazione di Numeri Pseudo-Casuali

Essendo macchine deterministiche, i computer non possono generare veri numeri casuali, ma utilizzano formule matematiche per produrre sequenze di numeri **pseudo-casuali**. Le funzioni per la casualità sono incluse nella libreria `<stdlib.h>`.

La funzione `rand()` e l'inizializzazione del seme (`srand`)

La funzione `rand()` restituisce un numero intero pseudo-casuale compreso tra 0 e la costante `RAND_MAX` (almeno 32767). Se eseguiamo un programma che usa `rand()` più volte, noteremo che genera sempre gli stessi identici numeri. Questo accade perché l'algoritmo parte da un valore di innesco predefinito, chiamato **seme (seed)**, impostato a 1.

Per ottenere numeri diversi ad ogni esecuzione, dobbiamo inizializzare il seme con un valore che cambia costantemente, come l'ora di sistema fornita dalla funzione `time(NULL)` della libreria `<time.h>`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int main(void) {
6     srand(time(NULL)); // Inizializza il seme con l'ora corrente (da
7                         // ⇨ invocare una sola volta)
8
9     printf("Numero casuale: %d\n", rand());
10    return 0;
11 }
```

Formula generale per l'intervallo $[a, b]$ (estremi inclusi)

Per ricondurre un numero casuale generato da `rand()` all'interno di un intervallo $[a, b]$, usiamo la formula:

```
1 int casuale = a + rand() % (b - a + 1);
```

Esempio: Sfida a dadi su 3 lanci

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int main(void) {
6     srand(time(NULL));
7
8     int somma_giocatore = 0;
9     int somma_computer = 0;
10
11    printf("=== SFIDA A DADI ===\n\n");
12 }
```

```

13     for (int i = 1; i <= 3; i++) {
14         int tiro_giocatore = 1 + rand() % 6; // Dado da 1 a 6
15         int tiro_computer = 1 + rand() % 6; // Dado da 1 a 6
16
17         printf("Lancio %d:\n", i);
18         printf("  Giocatore: %d\n", tiro_giocatore);
19         printf("  Computer:  %d\n\n", tiro_computer);
20
21         somma_giocatore += tiro_giocatore;
22         somma_computer += tiro_computer;
23     }
24
25     printf("Totale Giocatore: %d | Totale Computer: %d\n",
26 ↪      somma_giocatore, somma_computer);
27
28     if (somma_giocatore > somma_computer) {
29         printf("Hai vinto!\n");
30     } else if (somma_computer > somma_giocatore) {
31         printf("Vince il computer.\n");
32     } else {
33         printf("Pareggio!\n");
34     }
35
36     return 0;
37 }

```

5.3.2 La Cultura del Debugging: LLDB da Terminale e VS Code

Il debugger è lo strumento fondamentale per ispezionare l'esecuzione passo-passo del programma e trovare gli errori (bug) senza procedere per tentativi.

Compilazione ed esecuzione in LLDB

Compiliamo il codice aggiungendo il flag `-g` per inserire i simboli di debug:

```
1 clang -g -Wall programma.c -o programma
```

Quindi avviamo il debugger:

```
1 lldb ./programma
```

Una volta aperto il prompt (`lldb`), utilizziamo i comandi principali:

- **b main**: imposta un punto di interruzione (breakpoint) all'inizio del `main`.
- **r (run)**: avvia l'esecuzione del programma.
- **n (next)**: esegue la riga corrente e passa alla successiva (*Step Over*).
- **s (step)**: entra all'interno della funzione chiamata nella riga (*Step In*).
- **v (frame variable)**: stampa a schermo tutte le variabili locali attive nello Stack.

- **p nome\{}_var**: mostra il valore corrente di una singola variabile.
- **c** (continue): riprende l'esecuzione normale fino al prossimo breakpoint.
- **q** (quit): esce da LLDB.

Debugging Grafico in VS Code

Utilizzando l'estensione **CodeLLDB** ed i file `.vscode/tasks.json` e `launch.json` configurati:

1. Posiziona il mouse a sinistra del numero di riga nel file `.c` e clicca per inserire un breakpoint (cerchio rosso).
2. Premi **F5** per avviare la sessione di debug.
3. Avanza nel flusso usando i pulsanti grafici in alto (Step Over, Step In) e controlla lo stato della memoria in tempo reale nel pannello **Variables** a sinistra.

Codice di esempio per la sessione di Debug

Questo programma simula un algoritmo reale: calcolare la somma di tutti i numeri primi compresi tra 2 a 15 (il cui risultato corretto atteso è $2 + 3 + 5 + 7 + 11 + 13 = 41$).

Tuttavia, contiene **tre insidiosi bug logici estremamente comuni e difficili da individuare a occhio** (mancata inizializzazione dell'accumulatore, un errore di resettaggio del flag ed un errore sui limiti dei cicli). Compilando ed eseguendo, il programma restituirà un valore casuale senza senso. Usando il debugger scopriremo dove si inceppa la logica.

```
1  #include <stdio.h>
2
3  int main(void) {
4      int limit = 15;
5      int somma;           // BUG 1: variabile accumulatore non inizializzata
                           ↪ a 0! Conterr? valore spazzatura (garbage value).
6      int is_prime = 1; // BUG 2: dichiarato qui, ma non resettato a 1 ad
                           ↪ ogni iterazione del ciclo!
7
8      printf("Calcolo la somma dei numeri primi fino a %d...\n", limit);
9
10     for (int i = 2; i <= limit; i++) {
11         // Verifichiamo se i ? primo dividendolo per tutti i numeri da 2
        ↪ a i
12         for (int j = 2; j <= i; j++) { // BUG 3: j <= i includer? anche i
        ↪ stesso, quindi i % i == 0 sar? sempre vero!
13             if (i % j == 0) {
14                 is_prime = 0;
15                 break;
16             }
17         }
18
19         if (is_prime) {
20             somma += i;
21             printf("Trovato primo: %d\n", i);
```

```

22     }
23 }
24
25 printf("La somma finale dei numeri primi ?: %d (Attesa: 41)\n", somma
↵ );
26 return 0;
27 }

```

Come procedere all'ispezione con il debugger:

1. **Esecuzione iniziale:** Compilando ed eseguendo il codice così com'è, si noterà che il programma termina stampando un valore incoerente o enorme (es. 1042779).
2. **Indagine su BUG 1:** Imposta un breakpoint all'inizio del `main` ed avvia il debug. Ispezionando il valore di `somma` prima del ciclo, vedrai che contiene un numero casuale sporco (*garbage value*) derivante da rimasugli nello Stack della RAM.
 - *Correzione:* È fondamentale inizializzare l'accumulatore a zero: `int somma = 0;`.
3. **Indagine su BUG 3:** Riavvia il debug dopo aver corretto la dichiarazione di `somma`. Avanza passo-passo (`next`) per la prima iterazione (`i = 2`). Entrando nel ciclo interno, vedrai che per `j = 2` la condizione `j <= i` è vera. Di conseguenza, si esegue `is[j] % j == 0` (ovvero `2 % 2 == 0`, che è vero) impostando immediatamente `is[j]_prime = 0`. Il numero 2 viene quindi erroneamente scartato.
 - *Correzione:* La divisibilità va testata solo per i divisori minori del numero stesso. Dobbiamo correggere la condizione del ciclo interno in `j < i`.
4. **Indagine su BUG 2:** Riavvia il debug dopo aver applicato la seconda correzione. Avanzando passo-passo, vedrai che `i = 2` e `i = 3` vengono ora identificati come primi e sommati. Arrivando a `i = 4` (non primo), il ciclo imposta giustamente `is[i]_prime = 0`. All'iterazione successiva per `i = 5` (primo), ispezionando il valore della variabile `is[i]_prime` all'ingresso del ciclo vedrai che essa è **rimasta a 0** (il valore dell'iterazione precedente!).
 - *Correzione:* Il flag `is[i]_prime` deve essere resettato a 1 **all'inizio di ogni iterazione del ciclo esterno** (dichiarandola o inizializzandola all'interno del corpo del primo `for`, non prima di esso).

5.4 Quiz di Autovalutazione

Quiz 1

Domanda

1. Qual è la differenza fondamentale tra un ciclo `while` e un ciclo `do-while`?

- A) Il `do-while` può contenere solo variabili intere.
- B) Il `do-while` esegue il corpo delle istruzioni almeno una volta prima di valutare la condizione.
- C) Il `while` è più veloce da eseguire a livello hardware.
- D) Il `do-while` non permette l'uso di `break`.

Soluzione
Risposta corretta: B
Spiegazione: Valutando la condizione in coda (post-condizione), il corpo del `do-while` viene sempre eseguito una prima volta, a prescindere dal fatto che la condizione sia vera o falsa.

Quiz 2

Domanda

2. Quale delle seguenti definizioni esprime esattamente il ciclo `for (int i = 0; i < 5; i++) {}` come ciclo `while` equivalente?

- A) `int i = 0; while (i < 5) {} i++; {}`
- B) `while (int i = 0; i < 5) {} i++; ... {}`
- C) `int i = 0; while (i < 5) {} i++; ... {}`
- D) `int i = 0; do {} ... {} while (i < 5);`

Soluzione
Risposta corretta: A
Spiegazione: L'inizializzazione avviene prima del ciclo, il controllo della condizione si effettua in cima ad ogni iterazione, e l'incremento viene eseguito come ultima istruzione all'interno del corpo.

Quiz 3

Domanda

3. Se dichiariamo una variabile `int temp = 5`; all'interno delle parentesi graffe di un blocco `if`, cosa succede se proviamo a leggerla dopo la parentesi graffa di chiusura dell'`if`?

- A) Il compilatore legge il valore 5.
- B) Il programma compila ma restituisce un valore casuale (garbage value).
- C) Si verifica un errore di compilazione poiché la variabile non è definita al di fuori del blocco.
- D) Il sistema operativo arresta il programma con un crash di runtime.

Soluzione**Risposta corretta: C**

Spiegazione: In C vige lo scope di blocco (Block Scope). Una variabile dichiarata all'interno di `{ }` esiste ed è visibile solo all'interno di quelle parentesi graffe.

Quiz 4**Domanda**

4. A cosa serve la porzione di memoria RAM denominata Stack durante l'esecuzione del programma?

- A) Ad archiviare in modo permanente i file su disco.
- B) A contenere le istruzioni assembly del processore.
- C) Ad allocare e rimuovere automaticamente le variabili locali delle funzioni e dei blocchi attivi.
- D) A memorizzare le variabili globali utilizzate in tutti i file.

Soluzione**Risposta corretta: C**

Spiegazione: Lo Stack è una memoria a pila veloce utilizzata per la gestione del ciclo di vita delle variabili automatiche locali: le variabili vengono impilate all'ingresso del blocco e rimosse all'uscita.

Quiz 5**Domanda**

5. Qual è il valore massimo teorico che può essere restituito dall'espressione `rand() % 10`?

- A) 10
- B) 9
- C) 0
- D) RAND_MAX

Soluzione**Risposta corretta: B**

Spiegazione: L'operatore modulo `%` restituisce i resti della divisione intera per 10, che possono variare esclusivamente da 0 a 9 inclusi.

Quiz 6

Domanda

6. Perché è necessario invocare la funzione `srand(time(NULL))` all'inizio di un programma che genera numeri casuali?

- A) Per rendere l'esecuzione più veloce.
- B) Perché altrimenti la funzione `rand()` restituirebbe sempre zero.
- C) Per impostare un seme iniziale dinamico legato al tempo corrente, evitando di generare la stessa sequenza di numeri a ogni avvio del programma.
- D) Per calibrare il clock hardware della CPU del computer.

Soluzione
Risposta corretta: C
Spiegazione: I computer usano generatori deterministici. Senza inizializzare il seme con un valore variabile (come il tempo corrente), l'algoritmo pseudo-casuale partirebbe sempre dallo stesso stato iniziale generando sempre la stessa sequenza.

Quiz 7

Domanda

7. Nel debugger LLDB, qual è il comando utilizzato per avanzare di una singola riga di codice senza scendere all'interno delle funzioni chiamate?

- A) `step`
- B) `run`
- C) `next`
- D) `continue`

Soluzione
Risposta corretta: C
Spiegazione: Il comando `next (o n)` esegue l'istruzione corrente interamente ed avanza alla riga successiva dello stack frame corrente (`Step Over`).

Quiz 8

Domanda

8. Nel debugger LLDB, a cosa serve il comando `frame variable (o v)`?

- A) A cambiare il file sorgente visualizzato.
- B) A mostrare i valori di tutte le variabili locali attive nello stack frame corrente.
- C) A modificare il valore di una variabile in tempo reale.

- D) A terminare l'esecuzione del programma.

Soluzione
Risposta corretta: B
Spiegazione: Il comando `frame variable` (abbreviato in `v`) stampa a schermo il nome e il valore corrente di tutte le variabili locali allocate nello Stack per la porzione di codice in esecuzione.

Quiz 9

Domanda

9. Quale flag deve essere passato al compilatore (Clang/GCC) in fase di compilazione per abilitare i simboli di debug utili a LLDB?

- A) `-Wall`
- B) `-O3`
- C) `-g`
- D) `-std=c99`

Soluzione
Risposta corretta: C
Spiegazione: Il flag `-g` dice al compilatore di inserire nell'eseguibile le tabelle dei simboli che associano gli indirizzi macchina delle istruzioni alle righe del file sorgente C.

Quiz 10

Domanda

10. Quale libreria deve essere inclusa per poter usare la funzione `time(NULL)`?

- A) `<stdlib.h>`
- B) `<stdio.h>`
- C) `<time.h>`
- D) `<math.h>`

Soluzione
Risposta corretta: C
Spiegazione: La libreria standard `<time.h>` contiene le definizioni e le funzioni per la manipolazione e l'acquisizione dell'ora e della data di sistema.

5.5 Esercizi Pratici

Esercizio 1: Conto alla rovescia

Domanda

Scrivete un programma che acquisisca un numero intero positivo N da tastiera e stampi un conto alla rovescia a partire da N fino a 0 (incluso) utilizzando un ciclo `for`.

Spiegazione dell'algoritmo: Inizializziamo il contatore del ciclo `for` al valore N . A ogni iterazione, decrementiamo i di una unità ($i--$) e continuiamo finché i è maggiore o uguale a zero.

```

1  #include <stdio.h>
2
3  int main(void) {
4      int n = 0;
5
6      printf("Inserisci un numero intero positivo: ");
7      if (scanf("%d", &n) != 1 || n < 0) {
8          printf("Inserimento non valido.\n");
9          return 1;
10     }
11     // Conto alla rovescia decrescente
12     for (int i = n; i >= 0; i--) {
13         printf("%d ", i);
14     }
15     printf("\n");
16     return 0;
17 }

```

Soluzione

Esercizio 2: Somma di numeri positivi

Domanda

Scrivete un programma che chieda ripetutamente all'utente di inserire numeri interi positivi, sommandoli tra loro. Il programma deve terminare non appena l'utente inserisce il numero 0. Utilizzate un ciclo `do-while` e gestite la validazione in modo da ignorare (e segnalare come errore) eventuali numeri negativi senza interrompere il ciclo.

Soluzione

```

1  #include <stdio.h>
2
3  int main(void) {
4      int numero;
5      int somma = 0;
6
7      printf("Inserisci una serie di numeri positivi da sommare (0
      ↪ per terminare):\n");
8
9      do {
10         printf("Inserisci numero: ");
11         if (scanf("%d", &numero) != 1) {
12             printf("Errore di lettura.\n");
13             return 1;
14         }
15         if (numero < 0) {
16             printf("Errore: inserire solo numeri positivi!\n");
17         } else {
18             somma += numero;
19         }
20     } while (numero != 0);
21
22     printf("La somma totale dei numeri validi inseriti ?: %d\n",
23           ↪ somma);
24     return 0;
25 }

```

Spiegazione dell'algoritmo:

Il ciclo do-while garantisce che venga eseguita almeno una lettura. Se il valore inserito è negativo, stampiamo un messaggio di errore e non lo aggiungiamo alla variabile somma. Se il valore è positivo (o uguale a 0), lo sommiamo. Il ciclo termina non appena viene rilevato lo 0.

Spiegazione dell'algoritmo:

Dopo aver validato l'input, utilizziamo un ciclo for che conta da 1 a 10 per moltiplicare il numero N ad ogni passo, stampando il risultato formattato.

```

1  #include <stdio.h>
2
3  int main(void) {
4      int n;
5
6      printf("Inserisci un numero da 1 a 10: ");
7      if (scanf("%d", &n) != 1 || n < 1 || n > 10) {
8          printf("Valore non valido.\n");
9          return 1;
10     }
11     printf("Tabellina del %d:\n", n);
12     for (int i = 1; i <= 10; i++) {
13         printf("%d x %d = %d\n", n, i, n * i);
14     }
15     return 0;
16 }
17
18 
```

Soluzione**Esercizio 4: Griglia quadrata di asterischi****Domanda**

Scrivete un programma che legga da tastiera un intero positivo L e stampi una griglia quadrata di asterischi di lato L (righe e colonne pari a L), utilizzando cicli for annidati.

Soluzione

```

1  #include <stdio.h>
2
3  int main(void) {
4      int lato;
5
6      printf("Inserisci il lato del quadrato: ");
7      if (scanf("%d", &lato) != 1 || lato <= 0) {
8          printf("Lato non valido.\n");
9          return 1;
10     }
11     // Ciclo esterno per le righe
12     for (int i = 0; i < lato; i++) {
13         // Ciclo interno per le colonne della riga corrente
14         for (int j = 0; j < lato; j++) {
15             printf("%*s", j < lato; j++);
16         }
17         printf("\n"); // A capo al termine di ciascuna riga
18     }
19 }
20
21 return 0;
22 }

```

Spiegazione dell'algoritmo:

Il ciclo esterno gestisce la creazione delle L righe del quadrato. Per ogni singola riga crea dall'esterno, il ciclo interno provvede a stampare esattamente L asterischi affiancati da uno spazio.

Esercizio 5: Cornice quadrata vuota di asterischi**Domanda**

Scrivete un programma che legga da tastiera un intero positivo L e stampi un quadrato "vuoto" di lato L (ossia stampando asterischi solo sui bordi esterni e spazi all'interno).

Spiegazione dell'algoritmo:

Identifichiamo due cicli per scansionare ogni cella del quadrato. Grazie alla struttura condizionale `if`, stampiamo un asterisco solo se la coordinata corrente si trova su un bordo esterno (riga 0, riga lato - 1, colonna 0 o colonna lato - 1). Altrimenti stampiamo spazi vuoti.

```

1  #include <stdio.h>
2
3  int main(void) {
4      int lato;
5
6      printf("Inserisci il lato (minimo 2): ");
7      if (scanf("%d", &lato) != 1 || lato < 2) {
8          printf("Lato non valido.\n");
9          return 1;
10     }
11     for (int i = 0; i < lato; i++) {
12         for (int j = 0; j < lato; j++) {
13             // Stampiamo l'asterisco solo se siamo sulla prima/
14             // ultima riga o colonna
15             if (i == 0 || i == lato - 1 || j == 0 || j == lato - 1)
16                 printf("* ");
17             else {
18                 printf(" "); // Due spazi per compensare "*"
19             }
20             printf("\n");
21         }
22     }
23     return 0;
24 }

```

Soluzione**Esercizio 6: Caccia al numero casuale****Domanda**

Scrivete un programma che generi continuamente numeri casuali interi compresi tra 1 e 50 e li stampi a schermo. Il ciclo deve interrompersi non appena viene generato il numero 42, stampando a schermo il numero totale di tentativi effettuati.

Soluzione

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int main(void) {
6      srand(time(NULL));
7
8      int casuale;
9      int tentativi = 0;
10
11     printf("Generazione di numeri casuali fino al 42...\n");
12
13     do {
14         casuale = 1 + rand() % 50; // Valore tra 1 e 50
15         tentativi++;
16         printf("Tentativo %d:\n", tentativi, casuale);
17     } while (casuale != 42);
18
19     printf("\nTrovato 42 in %d tentativi\n", tentativi);
20     return 0;
21 }

```

Spiegazione dell'algoritmo:

Inizializziamo il seme dei numeri casuali con `time(NULL)`. Utilizziamo un ciclo `do-while` per generare un numero ad ogni iterazione, incrementando una variabile contatore. La condizione di controllo fa sì che il ciclo si ripeta finché il numero estratto è diverso da 42.

Esercizio 7: Tabella di potenze**Domanda**

Scrivete un programma che acquisisca un numero intero positivo N e visualizzi una tabella formattata a tre colonne contenente: il numero i , il suo quadrato i^2 , e il suo cubo i^3 , per tutti i valori di i da 1 a N .

Spiegazione dell'algoritmo: Utilizziamo un ciclo `for` da 1 a N . Per formattare le colonne in modo che i dati siano allineati a destra, sfruttiamo le specifiche di ampiezza minima di campo di `printf` (es. `\{ %10d` riserva almeno 10 caratteri di spazio per la stampa del numero).

```

1  #include <stdio.h>
2
3  int main(void) {
4      int n;
5
6      printf("Inserisci un intero positivo N: ");
7      if (scanf("%d", &n) != 1 || n <= 0) {
8          printf("Inserimento errato.\n");
9          return 1;
10     }
11     printf("%5s | %10s | %10s\n", "N", "Quadrato", "Cubo");
12     printf("-----\n");
13     for (int i = 1; i <= n; i++) {
14         int quadrato = i * i;
15         int cubo = i * i * i;
16         printf("%5d | %10d | %10d\n", i, quadrato, cubo);
17     }
18     return 0;
19 }
20
21
22

```

Soluzione

Esercizio 8: Calcolo della potenza (Moltiplicazione

Domanda

cumulativa)

Scrivete un programma che legga da tastiera due interi positivi: la base B e l'esponente E . Calcolate la potenza B^E utilizzando un ciclo, senza fare uso della funzione `pow` della libreria `<math.h>`.

base.

inizializziamo il risultato a 1. Eseguiamo un ciclo `for` che si ripete un numero di volte pari all'esponente, moltiplicando ad ogni iterazione il risultato accumulato per la

Spiegazione dell'algoritmo:

```

1  #include <stdio.h>
2
3  int main(void) {
4      int base, esponente;
5      long risultato = 1;
6
7      printf("Inserisci la base (B >= 0): ");
8      if (scanf("%d", &base) != 1 || base < 0) {
9          return 1;
10     }
11
12     printf("Inserisci l'esponente (E >= 0): ");
13     if (scanf("%d", &esponente) != 1 || esponente < 0) {
14         return 1;
15     }
16
17     // Moltiplicazione ripetuta
18     for (int i = 0; i < esponente; i++) {
19         risultato = risultato * base;
20     }
21
22     printf("%d elevato a %d ? part a: %ld\n", base, esponente,
23           risultato);
24     return 0;
25 }
```

Soluzione**Esercizio 9: Piramide di numeri****Domanda**

Scrivete un programma che acquisisca un numero intero positivo H (altezza) e stampi una piramide di numeri di altezza H . Ad esempio, per $H = 4$, il programma dovrà stampare:

```

1      1
2     2 2
3    3 3 3
4   4 4 4 4
```

Spiegazione dell'algoritmo: Utilizziamo un ciclo esterno che controlla le righe (da 1 a altezza). Per ciascuna riga i , stampiamo prima altezza - i spazi vuoti per spingere la piramide a destra, e poi eseguiamo un ciclo interno che stampa il valore di i (ossia il numero di riga corrente) per i volte.

```

1  #include <stdio.h>
2
3  int main(void) {
4      int altezza;
5
6      printf("Inserisci l'altezza della piramide: ");
7      if (scanf("%d", &altezza) != 1 || altezza <= 0) {
8          printf("Altezza non valida.\n");
9          return 1;
10     }
11
12     for (int i = 1; i <= altezza; i++) {
13         // 1. Stampa degli spazi iniziali per centrare la riga
14         for (int spazi = 0; spazi < altezza - i; spazi++) {
15             printf(" ");
16         }
17         // 2. Stampa dei numeri separati da spazio
18         for (int j = 0; j < i; j++) {
19             printf("%d ", i);
20         }
21         printf("\n"); // A capo a fine riga
22     }
23
24     }
25
26     return 0;
27 }
```

Soluzione

Esercizio 10: Simulatore di Tiro al Bersaglio (Gioco

Domanda

interattivo)

Scrivete un programma che simuli un gioco di tiro al bersaglio. Il bersaglio ha punteggi da 1 a 10. Ad ogni turno, il programma genera un tiro casuale (un valore da 1 a 10). Se il tiro è inferiore o uguale a 3, viene considerato un "mancato" e non assegna punti. Il gioco termina non appena la somma totale dei punti validi raggiunge o supera 30, oppure se il giocatore compie un totale di 8 tiri complessivi. Stampate l'esito di ogni tiro, il punteggio totale ed un messaggio di vittoria (se si raggiungono i 30 punti) o di sconfitta.

Spiegazione dell'algoritmo:

Il ciclo continua finché si verificano contemporaneamente due condizioni: il numero di tiri è inferiore al limite di 8, e il punteggio accumulato è inferiore all'obiettivo di 30. A ogni iterazione, generiamo un punteggio casuale, incrementiamo i tiri e controlliamo se il colpo è andato a segno (valore > 3). Terminati i cicli, valutiamo se la fine è avvenuta per superamento dell'obiettivo (Vittoria) o per esaurimento tiri (Sconfitta).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int main(void) {
6      srand(time(NULL));
7
8      int punteggio_totale = 0;
9      int tiri = 0;
10     int bersaglio;
11
12     printf("=== BERSAGLIO CASUALE ===\n");
13     printf("Obiettivo: Raggiungere 30 punti. Massimo 8 tentativi.\n");
14     ↵ "\n");
15     while (tiri < 8 && punteggio_totale < 30) {
16         bersaglio = 1 + rand() % 10; // Genera punteggio da 1 a 10
17         tiri++;
18         printf("Tiro %d: Colpo settore %d", tiri, bersaglio);
19
20         if (bersaglio <= 3) {
21             printf("<--> MANCATO! (0 punti)\n");
22         } else {
23             punteggio_totale += bersaglio;
24             printf("<--> Centro! (%d punti aggiunti)\n", bersaglio);
25         }
26         printf("Punteggio corrente: %d\n\n", punteggio_totale);
27     }
28     printf("Partita Terminata ---\n");
29     if (punteggio_totale >= 30) {
30         printf("Complimenti, hai vinto in %d tiri!\n", tiri);
31     } else {
32         printf("Tentativi esauriti! Hai perso. Punteggio finale: %d\n",
33             ↵ "\n", punteggio_totale);
34     }
35     return 0;
36 }
37
38
39

```

LEZIONE 6

Array Monodimensionali Statici e Algoritmi Base



Dispense di Programmazione 1 – Canale 0-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 6

Array Monodimensionali Statici e Algoritmi Base

Nelle lezioni precedenti abbiamo imparato a gestire variabili scalari, cicli e strutture di controllo. Tuttavia, ogni valore elaborato veniva subito “dimenticato”: con il `while` di Lezione 4, per esempio, non era possibile tornare indietro a rileggere un dato già processato. Questa lezione introduce il costrutto che risolve il problema: l'**array monodimensionale statico**. Un array è una sequenza di variabili dello stesso tipo allocate in modo contiguo in memoria, accessibili tramite un indice numerico. Impareremo a dichiararlo, inizializzarlo e a scrivere i tre algoritmi fondamentali che si applicano a qualsiasi sequenza di dati: calcolo della media, ricerca lineare e individuazione di massimo e minimo.

6.1 Motivazione e Anatomia degli Array

6.1.1 Il Problema che Motiva l'Array

Supponiamo di voler leggere i voti di 5 esami e calcolarne la media. Con le variabili scalari che conosciamo, saremmo costretti a scrivere:

```
1 int voto1, voto2, voto3, voto4, voto5;
2 scanf("%d", &voto1);
3 scanf("%d", &voto2);
4 // ... e cos? via per ogni voto
```

Questo approccio è insostenibile per 100 esami, e soprattutto è impossibile rielaborare i dati: una volta calcolata la media, non possiamo “tornare indietro” a scoprire, ad esempio, quali voti la superano.

L'**array** risolve entrambi i problemi: raggruppa N variabili dello stesso tipo in un'unica struttura indicizzata, e le mantiene tutte in memoria per tutto il tempo necessario.

6.1.2 Dichiarazione e Layout in RAM

La sintassi di dichiarazione è:

```
1 tipo nome_array[dimensione];
```

Esempi pratici:

```
1 int voti[5];           // 5 interi contigui in RAM (5 x 4 byte = 20 byte)
2 double misure[10];    // 10 double contigui in RAM (10 x 8 byte = 80 byte)
3 char lettere[26];     // 26 char contigui in RAM (26 x 1 byte = 26 byte)
```

Le celle di un array sono allocate in modo **contiguo** nello Stack — la zona di memoria temporanea riservata dal sistema operativo per le variabili locali della funzione corrente. Grazie alla contiguità, la CPU può calcolare l'indirizzo di qualsiasi elemento in tempo costante, senza dover scorrere la memoria.

Ogni cella ha un proprio **indice** che parte sempre da 0 e arriva a N-1:

Tabella

Indice	0	1	2	3	4
Elemento	voti[0]	voti[1]	voti[2]	voti[3]	voti[4]
Indirizzo (fittizio)	1000	1004	1008	1012	1016

Gli indirizzi dell'ultima riga sono fittizi, ma mostrano come vengono salvati i dati: ogni `int` occupa 4 byte, quindi le celle si trovano una dopo l'altra, a distanza di 4 byte l'una dall'altra.

Nota

Perché gli indici partono da 0? L'indice è in realtà un **offset** (scostamento) rispetto all'indirizzo base dell'array. Il primo elemento si trova esattamente all'indirizzo base, quindi il suo scostamento è 0. Questo diventerà ancora più chiaro quando studieremo i puntatori.

Attenzione

Out-of-Bounds: L'Errore Silenzioso Accedere a `voti[5]` su un array di 5 elementi (indici validi: 0-4) è un **comportamento non definito** (*Undefined Behavior*) in C. Il compilatore **non** segnala alcun errore: il programma legge o scrive silenziosamente in una zona di memoria che appartiene ad un'altra variabile, con risultati imprevedibili. È uno degli errori più pericolosi e frequenti in C.

6.1.3 Inizializzazione e Uso di `#define N`

Un array può essere inizializzato alla dichiarazione tramite una lista di valori tra parentesi graffe:

```
1 int voti[5] = {28, 30, 25, 27, 18}; // inizializzazione completa
2 int zeri[5] = {0};                 // tutti gli elementi inizializzati
   ⇨ a 0
3 int parziale[5] = {10, 20};        // voti[2], [3], [4] vengono posti
   ⇨ automaticamente a 0
```


Attenzione

Array non inizializzato: Garbage Value Se un array viene dichiarato senza inizializzazione, le sue celle contengono **valori spazzatura** (*garbage value*): i byte residui rimasti nello Stack dalle esecuzioni precedenti, del tutto casuali e imprevedibili.

```

1  #include <stdio.h>
2  #define N 5
3
4  int main(void) {
5      int voti[N]; // nessuna inizializzazione!
6
7      // SBAGLIATO: stampiamo i valori prima di assegnarli
8      for (int i = 0; i < N; i++) {
9          printf("voti[%d] = %d\n", i, voti[i]); // output
          ↪ imprevedibile!
10     }
11     return 0;
12 }
```

L'output di questo programma sarà completamente casuale — potrebbe essere 0 (se lo Stack è stato azzerato dal sistema operativo), oppure numeri enormi come 32767, -858993460 o qualsiasi altro residuo di memoria. Questo comportamento è **non definito** dallo standard C, il che significa che il compilatore non è obbligato a produrre un risultato coerente.

Come evitarlo: Inizializza sempre un array prima di usarlo, o con una lista esplicita, o con `\{\}\{0\}` per azzerarlo, o leggendo i valori dall'utente con `scanf` prima di accedervi.

```

1  int voti[5] = {0}; // corretto: tutti gli elementi a 0
2  int voti[5] = {28, 30, 25, 27, 18}; // corretto: valori espliciti
```

Quando rilevante, è buona pratica definire la dimensione dell'array come costante simbolica tramite `\{\}\#define`, così da poterla modificare in un solo punto:

```

1  #define N 5 // cambiando questo numero, si aggiorna tutto il codice che
    ↪ usa N
2
3  int main(void) {
4      int voti[N];
5      // ...
6  }
```

L'accesso ai singoli elementi avviene tramite l'operatore di indicizzazione `[]`:

```

1  voti[0] = 28; // scrittura: assegna 28 al primo elemento
2  printf("%d\n", voti[2]); // lettura: stampa il terzo elemento (indice 2)
```

6.1.4 La dimensione di un Array: L'operatore sizeof

L'operatore `sizeof` restituisce lo spazio occupato in memoria da un tipo di dato o da una variabile, espresso in **byte**. Quando viene applicato a un array statico, `sizeof` restituisce la **dimensione totale in byte** dell'intero array, non il numero di elementi.

Ad esempio, se abbiamo un array di 5 interi:

```
1 int voti[5];
2 printf("Spazio occupato: %zu byte\n", sizeof(voti)); // Stampa 20 (5
    ↪ elementi * 4 byte ciascuno)
```

Per determinare in modo automatico il **numero di elementi** (la lunghezza dell'array) senza cablare la costante nel codice, si divide la dimensione totale dell'array per la dimensione di un singolo elemento (solitamente il primo elemento a indice 0):

```
1 int lunghezza = sizeof(voti) / sizeof(voti[0]);
2 // Equivale a: 20 byte / 4 byte = 5 elementi
```

Questa espressione viene calcolata a compile-time (durante la compilazione) ed è estremamente comoda per scrivere cicli generici in cui non vogliamo hardcodare la costante numerica:

```
1 for (int i = 0; i < (sizeof(voti) / sizeof(voti[0])); i++) {
2     // codice di scansione
3 }
```

Attenzione

Attenzione al ciclo di vita della dimensione L'operatore `sizeof` restituisce la dimensione corretta dell'array **solo** all'interno dello stesso scope (blocco di codice) in cui l'array è stato dichiarato. Se l'array viene passato come parametro a una funzione, decade a un semplice puntatore (Array Decay) e `sizeof` restituirà la dimensione del puntatore (8 byte su sistemi a 64-bit), rendendo impossibile risalire alla lunghezza originaria in questo modo.

6.2 Il Pattern di Scansione e gli Algoritmi Base

6.2.1 Il Ciclo for come Strumento Universale sugli Array

Quasi tutti gli algoritmi su array condividono lo stesso schema mentale: un ciclo `for` che scorre tutti gli indici da 0 a N-1.

```
1 for (int i = 0; i < N; i++) {
2     // opera su arr[i]
3 }
```

Questo pattern è così ricorrente che vale la pena memorizzarlo come un blocco atomico di pensiero. Vediamo come usarlo per leggere e stampare un array:

```

1  #include <stdio.h>
2  #define N 5
3
4  int main(void) {
5      int voti[N];
6
7      // Lettura dell'array
8      printf("Inserisci %d voti:\n", N);
9      for (int i = 0; i < N; i++) {
10         printf("Voto %d: ", i + 1); // i+1 perche l'utente conta da 1
11         scanf("%d", &voti[i]);
12     }
13
14     // Stampa dell'array
15     printf("\nVoti inseriti: ");
16     for (int i = 0; i < N; i++) {
17         printf("%d ", voti[i]);
18     }
19     printf("\n");
20
21     return 0;
22 }

```

Si noti che usiamo **due cicli distinti**: uno per la lettura, uno per la stampa. Con un array possiamo ripercorrere i dati tutte le volte che vogliamo – cosa impossibile con variabili scalari.

6.2.2 Algoritmo: Calcolo della Media

La media aritmetica si calcola accumulando la somma in un unico ciclo e dividendo una sola volta alla fine.

```

1  #include <stdio.h>
2  #define N 5
3
4  int main(void) {
5      int voti[N] = {28, 30, 25, 27, 18};
6      int somma = 0;
7
8      for (int i = 0; i < N; i++) {
9          somma += voti[i];
10     }
11
12     double media = (double)somma / N; // cast esplicito per la divisione
    ⇨ reale!
13     printf("Media: %.2f\n", media);
14
15     return 0;
16 }

```

Attenzione

Il cast (double) è obbligatorio. Senza il cast esplicito, `somma / N` sarebbe una divisione intera che tronca il risultato decimale. Ad esempio, `128 / 5` darebbe 25 invece di 25.60. Il cast (double) `somma` converte il dividendo in virgola mobile prima della divisione.

6.2.3 La Prima Vera Dimostrazione del Valore degli Array: la Seconda Passata

Dopo aver calcolato la media, possiamo fare una **seconda passata** sull'array per scoprire quali voti la superano. Questa operazione sarebbe **impossibile** senza un array: con variabili scalari, ogni valore inserito dall'utente viene scartato dopo l'elaborazione e non può essere riletto.

```
1 // ... dopo aver calcolato 'media' con il ciclo precedente ...
2
3 printf("\nVoti superiori alla media (%.2f):\n", media);
4 for (int i = 0; i < N; i++) {
5     if (voti[i] > media) {
6         printf(" - voti[%d] = %d\n", i, voti[i]);
7     }
8 }
```

Il confronto `voti[i] > media` funziona correttamente: poiché `media` è un `double`, il compilatore promuove automaticamente `voti[i]` (intero) a `double` prima del confronto.

6.3 Algoritmi di Ricerca e di Estremi

6.3.1 Algoritmo: Ricerca Lineare (Sequential Search)

La ricerca lineare è l'algoritmo più semplice per trovare un elemento in un array non ordinato: si esaminano le celle una per una dall'inizio, finché non si trova l'elemento cercato o si esaurisce l'array.

Esistono due varianti comuni:

Pattern A: Determinare SE l'elemento esiste (flag booleano)

```
1 #define N 5
2 int voti[N] = {28, 30, 25, 27, 18};
3 int target = 30;
4 int trovato = 0; // flag: 0 = non trovato, 1 = trovato
5
6 for (int i = 0; i < N; i++) {
7     if (voti[i] == target) {
8         trovato = 1;
9         break; // inutile continuare una volta trovato
10    }
11 }
```

```

12
13 if (trovato) {
14     printf("Valore %d trovato nell'array.\n", target);
15 } else {
16     printf("Valore %d non presente.\n", target);
17 }

```

Pattern B: Determinare DOVE si trova (indice sentinella)

```

1 int indice = -1; // -1 indica "non trovato" (nessun indice valido e'
    ↪ negativo)
2
3 for (int i = 0; i < N; i++) {
4     if (voti[i] == target) {
5         indice = i;
6         break;
7     }
8 }
9
10 if (indice != -1) {
11     printf("Valore %d trovato alla posizione %d.\n", target, indice);
12 } else {
13     printf("Valore %d non presente.\n", target);
14 }

```

Nota

Perché usare -1 come sentinella? Gli indici validi di un array sono sempre ≥ 0 . Il valore -1 è quindi impossibile come indice reale e viene usato universalmente in C come segnale di "non trovato". Il `break` è importante per l'efficienza: una volta trovato l'elemento, non ha senso continuare la scansione.

6.3.2 Algoritmo: Massimo e Minimo su Array

L'algoritmo per trovare il massimo e il minimo di un array segue la stessa logica già vista in Lezione 4 per la sequenza fissa: si inizializzano le variabili di riferimento con il **primo elemento** dell'array (non con 0 o un valore arbitrario), poi si scorre il resto dell'array dal secondo elemento in poi.

```

1 #include <stdio.h>
2 #define N 5
3
4 int main(void) {
5     int voti[N] = {28, 30, 25, 27, 18};
6
7     int max = voti[0]; // inizializzazione corretta: primo elemento
8     int min = voti[0];
9     int pos_max = 0; // teniamo anche l'indice della posizione
10    int pos_min = 0;
11

```

```
12  for (int i = 1; i < N; i++) { // partiamo da 1, non da 0!
13      if (voti[i] > max) {
14          max = voti[i];
15          pos_max = i;
16      }
17      if (voti[i] < min) {
18          min = voti[i];
19          pos_min = i;
20      }
21  }
22
23  printf("Voto massimo: %d (posizione %d)\n", max, pos_max);
24  printf("Voto minimo: %d (posizione %d)\n", min, pos_min);
25
26  return 0;
27 }
```

Nota

Perché il ciclo parte da $i = 1$? L'elemento `voti[0]` è già stato usato come valore di partenza per `max` e `min`. Confrontarlo con sé stesso sarebbe corretto ma ridondante. Il ciclo parte da $i = 1$ per confrontare tutti i rimanenti elementi con il riferimento iniziale.

6.3.3 Esempio Integrato: Analisi Statistica dei Voti

Quest'ultimo esempio mette insieme tutti gli algoritmi visti: leggiamo N voti validati da tastiera, calcoliamo media, massimo e minimo, e contiamo i voti superiori alla media.

```
1  #include <stdio.h>
2  #define N 5
3
4  int main(void) {
5      int voti[N];
6      int somma = 0;
7
8      // Acquisizione con validazione do-while (ripasso Lezione 5)
9      printf("Inserisci %d voti (18-30):\n", N);
10     for (int i = 0; i < N; i++) {
11         do {
12             printf("Voto %d: ", i + 1);
13             scanf("%d", &voti[i]);
14         } while (voti[i] < 18 || voti[i] > 30);
15         somma += voti[i];
16     }
17
18     // Calcolo media
19     double media = (double)somma / N;
20
21     // Calcolo max, min e conteggio in un unico ciclo
22     int max = voti[0], min = voti[0];
23     int superano_media = 0;
24     for (int i = 0; i < N; i++) {
```

```

25     if (voti[i] > max) max = voti[i];
26     if (voti[i] < min) min = voti[i];
27     if (voti[i] > media) superano_media++;
28 }
29
30 // Report finale
31 printf("\n--- Analisi Voti ---\n");
32 printf("Media:           %.2f\n", media);
33 printf("Massimo:          %d\n", max);
34 printf("Minimo:           %d\n", min);
35 printf("Sopra la media: %d/%d\n", superano_media, N);
36
37 return 0;
38 }

```

6.4 Quiz di Autovalutazione

Quiz 1

Domanda

1. Qual è il primo indice valido di un array in C?

- A) 1
- B) -1
- C) 0
- D) Dipende dal tipo dell'array

In C gli indici degli array partono sempre da 0. L'ultimo elemento valido di un array di N elementi si trova all'indice N-1.

Risposta corretta: C
Soluzione

Quiz 2

Domanda

2. Se dichiaro `int arr[8];`, quali sono gli indici validi?

- A) Da 0 a 8
- B) Da 1 a 8
- C) Da 0 a 7
- D) Da 1 a 7

Un array di dimensione N ha indici validi da 0 a $N-1$. Per `arr[8]`, gli indici validi sono da 0 a 7 . Accedere ad `arr[8]` è un accesso fuori dai limiti (out-of-bounds).

Risposta corretta: C

Soluzione

Quiz 3

Domanda

3. Cosa produce il codice `int a[3] = \{\}\{0\}\{\}\};`?

- A) Errore di compilazione: mancano tutti i valori.
- B) Un array di 3 elementi, tutti inizializzati a 0.
- C) Un array di 3 elementi, solo il primo a 0, gli altri con garbage value.
- D) Un array di 1 elemento con il valore 0.

In C, quando si fornisce un iniziatore parziale, tutti gli elementi non specificati vengono automaticamente a 0. \{\}\} è un modo comune e corretto per azzerare un intero array.

Risposta corretta: B

Soluzione

Quiz 4

Domanda

4. Perché è necessario il cast (double) somma nell'espressione (double) somma / N?

- A) Per aumentare la velocità di esecuzione.
- B) Per evitare l'overflow del tipo `int`.
- C) Per forzare una divisione in virgola mobile invece che una divisione intera che tronca il risultato.
- D) Perché `N` è sempre un `double`.

Senza il cast, somma / N sarebbe una divisione intera. Per esempio, 128 / 5 darebbe 25 invece di 25.6. Il cast (double) somma converte il dividendo in double prima della divisione, forzando un'operazione in virgola mobile.

Risposta corretta: C

Soluzione

Quiz 5

Domanda

5. Quale valore si usa convenzionalmente per inizializzare indice a "elemento non trovato" nella ricerca lineare?

- A) 0
- B) N (la dimensione dell'array)
- C) -1
- D) `INT\{}\_MAX`

Il valore -1 è la convenzione universale per indicare "non trovato" negli algoritmi di ricerca in C. Poiché gli indici validi sono sempre ≥ 0 , il valore -1 non può mai essere un indice reale e serve come sentinella inequivocabile.

Risposta corretta: C

Soluzione

Quiz 6

Domanda

6. Come si deve inizializzare la variabile `max` nell'algoritmo per trovare il massimo di un array?

- A) A 0, per confrontare valori positivi.
- B) Al valore più grande possibile (`INT\{}\_MAX`).
- C) Al valore del primo elemento dell'array (`arr[0]`).
- D) Non importa, qualsiasi valore va bene.

L'inizializzazione corretta è sempre il primo elemento dell'array. Inizializzare a 0 sarebbe sbagliato se tutti gli elementi fossero negativi. L'unica strategia sempre corretta è usare il primo elemento come valore di partenza.

Risposta corretta: C

Soluzione

Quiz 7

Domanda

7. Nel ciclo `for (int i = 1; i < N; i++)` per trovare il massimo, perché si parte da `i = 1`?

- A) Per evitare un accesso fuori dai limiti.
- B) Perché `arr[0]` è già usato come valore iniziale di `max`: confrontarlo di nuovo sarebbe ridondante.

- C) Perché il primo elemento non può mai essere il massimo.
- D) Per una convenzione del linguaggio C.

Soluzione
Risposta corretta: B
 Poiché abbiamo già assegnato `max = arr[0]`, il primo elemento ha già "concorso" come candidato. Iniziare il ciclo di confronto da `i = 1` evita un confronto inutile dell'elemento con se stesso.

Quiz 8

Domanda

8. Cosa accade se si accede a un indice fuori dai limiti in C (es. `arr[N]` su un array di `N` elementi)?

- A) Il compilatore genera sempre un errore che blocca la compilazione.
- B) Il programma si arresta immediatamente con un messaggio di errore chiaro.
- C) Il comportamento è non definito (Undefined Behavior): il programma potrebbe leggere dati casuali, corrompere la memoria o crashare.
- D) Il programma ritorna automaticamente all'ultimo elemento valido.

Soluzione
Risposta corretta: C
 L'accesso out-of-bounds è uno dei comportamenti non definiti più pericolosi del C. Il compilatore non garantisce alcun controllo a run-time sugli indici. Il programma potrebbe sembrare funzionare correttamente in alcuni casi e crashare in altri.

Quiz 9

Domanda

9. Qual è il vantaggio di usare `\{\}\#define N 5` rispetto a usare direttamente `5` nel codice?

- A) Riduce il tempo di compilazione.
- B) Permette di modificare la dimensione in un solo posto, aggiornando automaticamente tutto il codice che usa `N`.
- C) Rende l'array allocato nello Heap invece che nello Stack.
- D) Non ci sono vantaggi, è solo una convenzione stilistica.

Soluzione**Risposta corretta: B**

Usare una costante simbolica è il principio del "numero magico unico": se la dimensione cambia, basta modificarla in un solo posto. Senza questa pratica, bisognerebbe trovare e modificare manualmente ogni occorrenza del numero 5, rischiando di dimenticarne qualcuna.

Quiz 10**Domanda****10. Quale affermazione sulla "seconda passata" su un array è corretta?**

- A) È impossibile scorrere un array due volte nello stesso programma.
- B) Un array permette di elaborare gli stessi dati con più cicli sequenziali, cosa impossibile con variabili scalari.
- C) La seconda passata è sempre più veloce della prima perché i dati sono in cache.
- D) La seconda passata richiede che l'array sia ordinato.

Soluzione**Risposta corretta: B**

Questo è il vantaggio fondamentale degli array rispetto alle variabili scalari: i dati rimangono in memoria per tutta la durata del programma. Un esempio pratico è la stampa dei voti superiori alla media: possiamo prima calcolare la media, poi fare una seconda passata per confrontare ogni elemento con essa — operazione impossibile se avessimo letto i voti uno alla volta senza memorizzarli.

6.5 Esercizi Pratici

Esercizio 1: Stampa dell'Array al Contrario**Domanda**

Scrivete un programma che legga N interi da tastiera (con `\#define N 6`) e li stampi nell'ordine inverso rispetto a quello di inserimento.

Per stampare l'array al contrario, il secondo ciclo for parte dall'ultimo indice valido (N - 1) e decrementa i fino a 0 incluso (i >= 0). Non è necessario creare un secondo array: basta percorrere il primo in senso inverso.

```

1  #include <stdio.h>
2  #define N 6
3
4  int main(void) {
5      int arr[N];
6
7      printf("Inserisci %d numeri:\n", N);
8      for (int i = 0; i < N; i++) {
9          printf("Numero %d: ", i + 1);
10         scanf("%d", &arr[i]);
11     }
12     printf("Array al contrario: ");
13     for (int i = N - 1; i >= 0; i--) {
14         printf("%d ", arr[i]);
15     }
16     printf("\n");
17     return 0;
18 }
19
20

```

Soluzione

Esercizio 2: Somma e Media

Domanda

Scrivete un programma che legga N interi (con `\{\}\#define N 8`) e stampi la loro somma e la loro media con 2 decimali.

Accumulo la somma durante il ciclo di lettura per evitare un ciclo extra. La media richiede il cast (double) su somma per ottenere la divisione in virgola mobile.

```

1  #include <stdio.h>
2  #define N 8
3
4  int main(void) {
5      int arr[N];
6      int somma = 0;
7
8      printf("Inserisci %d numeri interi:\n", N);
9      for (int i = 0; i < N; i++) {
10         printf("Numero %d: ", i + 1);
11         scanf("%d", &arr[i]);
12         somma += arr[i];
13     }
14     double media = (double)somma / N;
15     printf("\nSomma: %d\n", somma);
16     printf("Media: %.2f\n", media);
17
18     return 0;
19 }
20

```

Soluzione

Esercizio 3: Contare gli Elementi Maggiori di una Soglia

Domanda

Scrivete un programma che legga N temperature in gradi Celsius (con `\{\}\#define N 7`) e conti quante superano 36.5. Usate un array di `double`.

Si noti l'uso di `\{\}\%lf` in `scanf` per leggere i valori `double`. La seconda passata con il ciclo di conteggio dimostra il valore dell'array: elaboriamo i dati separatamente rispetto alla lettura.

```

1  #include <stdio.h>
2  #define N 7
3
4  int main(void) {
5      double temp[N];
6      int febre = 0;
7
8      printf("Inserisci %d temperature:\n", N);
9      for (int i = 0; i < N; i++) {
10         printf("Temperatura %d: ", i + 1);
11         scanf("%lf", &temp[i]);
12     }
13
14     for (int i = 0; i < N; i++) {
15         if (temp[i] > 36.5) {
16             febre++;
17         }
18     }
19
20     printf("\nTemperature sopra 36.5: %d su %d\n", febre, N);
21
22     return 0;
23 }
```

Soluzione

Esercizio 4: Massimo e sua Posizione

Domanda

Scrivete un programma che legga `N` interi (con `\{\}\#define N 6`) e stampi il valore massimo e l'indice in cui si trova.

max viene inizializzato al primo elemento (arr[0]) e pos\{}_max a 0. Il ciclo parte da i = 1. Ogni volta che troviamo un elemento maggiore dell'attuale massimo, aggiorniamo sia il valore che la posizione.

```

1  #include <stdio.h>
2  #define N 6
3
4  int main(void) {
5      int arr[N];
6
7      printf("Inserisci %d numeri:\n", N);
8      for (int i = 0; i < N; i++) {
9          printf("Numero %d: ", i + 1);
10         scanf("%d", &arr[i]);
11     }
12     int max = arr[0];
13     int pos_max = 0;
14
15     for (int i = 1; i < N; i++) {
16         if (arr[i] > max) {
17             max = arr[i];
18             pos_max = i;
19         }
20     }
21
22     printf("Massimo: %d (indice %d)\n", max, pos_max);
23
24     return 0;
25 }

```

Soluzione

Esercizio 5: Ricerca Lineare con Indice

Domanda

Scrivete un programma che legga N interi (con \{}\#define N 5), chieda un valore da cercare e stampi l'indice della prima occorrenza, oppure un messaggio se non è presente.

La variabile `indice` viene inizializzata a `-1` come sentinella. Il `break` interrompe la ricerca appena l'elemento viene trovato, evitando scansioni inutili.

```

1  #include <stdio.h>
2  #define N 5
3
4  int main(void) {
5      int arr[N];
6
7      printf("Inserisci %d numeri:\n", N);
8      for (int i = 0; i < N; i++) {
9          printf("Numero %d: ", i + 1);
10         scanf("%d", &arr[i]);
11     }
12     int target;
13     printf("Numero da cercare: ");
14     scanf("%d", &target);
15
16     int indice = -1;
17     for (int i = 0; i < N; i++) {
18         if (arr[i] == target) {
19             indice = i;
20             break;
21         }
22     }
23     if (indice != -1) {
24         printf("trovato all'indice %d.\n", indice);
25     } else {
26         printf("non presente nell'array.\n", target);
27     }
28     return 0;
29 }

```

Soluzione

Esercizio 6: Minimo e Massimo Simultanei

Domanda

Scrivete un programma che legga `N` interi (con `\{\}#define N 7`) e stampi sia il valore minimo che il massimo in un unico ciclo di scansione.

separate.

Entrambe le variabili vengono inizializzate ad `arr[0]`. Nello stesso ciclo `for`, aggiorniamo massimo e minimo con due `if` indipendenti, evitando due passate

```

1  #include <stdio.h>
2  #define N 7
3
4  int main(void) {
5      int arr[N];
6
7      printf("Inserisci %d numeri:\n", N);
8      for (int i = 0; i < N; i++) {
9          printf("Numero %d: ", i + 1);
10         scanf("%d", &arr[i]);
11     }
12     int max = arr[0];
13     int min = arr[0];
14
15     for (int i = 1; i < N; i++) {
16         if (arr[i] > max) max = arr[i];
17         if (arr[i] < min) min = arr[i];
18     }
19
20     printf("Massimo: %d\n", max);
21     printf("Minimo: %d\n", min);
22
23     return 0;
24 }
25

```

Soluzione

Esercizio 7: Voti Superiori alla Media

Domanda

Scrivete un programma che legga N voti interi (con `\#\define N 6`), calcoli la media e stampi gli indici e i valori dei voti che la superano.

Questo esercizio esemplifica la "seconda passata": prima calcoliamo la media, poi scorriamo nuovamente l'array per trovare gli elementi che la superano.

```

1  #include <stdio.h>
2  #define N 6
3
4  int main(void) {
5      int voti[N];
6      int somma = 0;
7
8      printf("Inserisci %d voti:\n", N);
9      for (int i = 0; i < N; i++) {
10         printf("Voto %d: ", i + 1);
11         scanf("%d", &voti[i]);
12         somma += voti[i];
13     }
14     double media = (double)somma / N;
15     printf("Media: %.2f\n", media);
16     printf("\nVoti superiori alla media:\n");
17     int trovati = 0;
18     for (int i = 0; i < N; i++) {
19         if (voti[i] > media) {
20             printf(" - %d\n", i, voti[i]);
21             trovati++;
22         }
23     }
24     if (trovati == 0) {
25         printf("Nessun voto superiore alla media.\n");
26     }
27     return 0;
28 }

```

Soluzione

Esercizio 8: Array con Inizializzazione e Stampa

Domanda

Formattata

Dichiarate e inizializzate un array di 5 interi con i valori `{12, 7, 25, 3, 18}`. Stampate ogni elemento con il formato `arr[i] = valore`, poi stampate la somma totale.

L'array viene inizializzato alla dichiarazione. Un unico ciclo gestisce sia la stampa formattata che l'accumulo della somma.

```

1  #include <stdio.h>
2  #define N 5
3
4  int main(void) {
5      int arr[N] = {12, 7, 25, 3, 18};
6      int somma = 0;
7
8      printf("Contenuto dell'array:\n");
9      for (int i = 0; i < N; i++) {
10         printf("arr[%d] = %d\n", i, arr[i]);
11         somma += arr[i];
12     }
13     printf("\nSomma totale: %d\n", somma);
14     return 0;
15 }

```

Soluzione

Esercizio 9: Contare le Occorrenze di un Valore

Domanda

Scrivete un programma che legga N interi (con `\{\}\#define N 8`), chieda un valore target e stampi quante volte compare nell'array.

A differenza della ricerca con **break**, qui **non** usciamo dal ciclo appena troviamo il valore: dobbiamo contare **tutte** le occorrenze. Il ciclo scorre l'intero array senza interruzioni.

```

1  #include <stdio.h>
2  #define N 8
3
4  int main(void) {
5      int arr[N];
6
7      printf("Inserisci %d numeri:\n", N);
8      for (int i = 0; i < N; i++) {
9          printf("Numero %d: ", i + 1);
10         scanf("%d", &arr[i]);
11     }
12     int target;
13     printf("Valore da contare: ");
14     scanf("%d", &target);
15     int contatore = 0;
16     for (int i = 0; i < N; i++) {
17         if (arr[i] == target) {
18             contatore++;
19         }
20     }
21     printf("%d compare %d volta/e nell'array.\n", target, contatore);
22 }
23
24 ↵
25 return 0;
26
27 }
```

Soluzione

Esercizio 10: Analisi Completa di Temperature Giornaliere

Domanda

Scrivete un programma che simuli l'analisi di N temperature giornaliere (con `\{\}\#define N 7`). Il programma deve:

1. Leggere le 7 temperature (double tra -50.0 e 60.0 , validate con `do-while`).
2. Calcolare e stampare la temperatura media settimanale.
3. Stampare il giorno con la temperatura massima e quello con la minima.
4. Contare e stampare quanti giorni hanno avuto temperatura superiore alla media.

Soluzione

Questo esercizio integra tutti i pattern della lezione: acquisizione validata con `while`, media come divisione `double`, `max/min` con tracciamento della posizione, e conteggio degli elementi superiori alla media in un'unica passata. Si noti che `pos\{\}_max + 1` viene usato nella stampa per convertire dall'indice base-0 al numero del giorno da 1 a 7, più naturale per l'utente.

```

1  #include <stdio.h>
2  #define N 7
3
4  int main(void) {
5      double temp[N];
6      double somma = 0.0;
7
8      printf("Inserisci le temperature dei %d giorni:\n", N);
9      for (int i = 0; i < N; i++) {
10         do {
11             printf("Giorno %d (gradi C): ", i + 1);
12             scanf("%lf", &temp[i]);
13             while (temp[i] > 60.0 || temp[i] < -50.0) {
14                 somma += temp[i];
15             }
16         } while (temp[i] > 60.0 || temp[i] < -50.0);
17         double media = somma / N;
18
19         double max = temp[0], min = temp[0];
20         int pos_max = 0, pos_min = 0;
21         int sopra_media = 0;
22
23         for (int i = 0; i < N; i++) {
24             if (temp[i] > max) { max = temp[i]; pos_max = i; }
25             if (temp[i] < min) { min = temp[i]; pos_min = i; }
26             if (temp[i] > media) sopra_media++;
27         }
28
29         printf("\n--- Analisi Settimanale ---\n");
30         printf("Media: %.2f gradi C\n", media);
31         printf("Massima: %.2f gradi C (Giorno %d)\n", max, pos_max + 1);
32         printf("Minima: %.2f gradi C (Giorno %d)\n", min, pos_min + 1);
33         printf("Giorni sopra media: %d/%d\n", sopra_media, N);
34
35         return 0;
36     }

```

LEZIONE 7

L'Astrazione del Codice e l'Anatomia dello Stack



Dispense di Programmazione 1 – Canale 0-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 7

L'Astrazione del Codice e l'Anatomia dello Stack

Nelle lezioni precedenti abbiamo imparato a strutturare programmi in grado di compiere scelte e ripetere istruzioni. Tuttavia, man mano che la complessità dei problemi aumenta, scrivere tutto il codice all'interno di un unico blocco monolitico (il `main`) porta inevitabilmente a codici confusi, difficili da leggere e pieni di duplicazioni (il pericoloso pattern del copia-incolla). Questa lezione introduce lo strumento fondamentale per superare questo limite: le **funzioni**. Impareremo a scomporre un problema complesso in sotto-problemi indipendenti, a definire l'interfaccia di una funzione e a comprendere cosa accade "sotto il cofano" della memoria RAM, analizzando la struttura dello **Stack** e l'evoluzione dei **Record di Attivazione**.

7.1 Il Limite del Codice Monolitico e il Principio DRY

7.1.1 Perché il copia-incolla è il male

Immaginiamo di dover trovare il valore massimo all'interno di tre array distinti nel nostro programma. Senza uno strumento per riutilizzare la logica di calcolo, l'unica soluzione possibile sarebbe duplicare lo stesso codice per tre volte, adattando manualmente i nomi delle variabili:

```
1 // Ricerca massimo nel primo array
2 int max1 = array1[0];
3 for (int i = 1; i < N; i++) {
4     if (array1[i] > max1) {
5         max1 = array1[i];
6     }
7 }
8
9 // Ricerca massimo nel secondo array (COPIA-INCOLLA)
10 int max2 = array2[0];
11 for (int i = 1; i < N; i++) {
12     if (array2[i] > max2) {
13         max2 = array2[i];
14     }
15 }
16
```

```
17 // E cos? via per il terzo array...
```

Quando ci troviamo a dover eseguire lo stesso calcolo su insiemi di dati diversi, la tentazione immediata è quella di copiare e incollare il blocco di codice interessato, modificando solo i nomi delle variabili. Questo approccio viola il principio cardine dell'Ingegneria del Software noto come **DRY (Don't Repeat Yourself)**.

Se in un blocco di codice duplicato è presente un bug (ad esempio un errore off-by-one nel limite di un ciclo), saremo costretti a rintracciare e correggere l'errore in ogni singola copia. Qualora dimenticassimo anche solo una di queste copie, il programma presenterebbe comportamenti inconsistenti.

7.1.2 L'approccio procedurale

Per risolvere questo problema, il linguaggio C adotta il paradigma **procedurale**. Esso permette di isolare una porzione di codice, assegnarle un nome auto-esplicativo e richiamarla ogni volta che ne sorge la necessità.

Questa operazione prende il nome di **astrazione**: l'utente della funzione ha bisogno di conoscere solo l'interfaccia (cosa entra e cosa esce) senza doversi preoccupare dei dettagli implementativi interni.

In realtà, abbiamo utilizzato le funzioni fin dal nostro primissimo programma: in C non è possibile scrivere codice senza ricorrere ad esse, dato che il `main` stesso è una funzione obbligatoria che funge da punto d'ingresso del programma.

Inoltre, abbiamo fatto largo uso delle funzioni della libreria standard (quali `printf`, `scanf`, `pow`, `sqrt`, `rand`, `srand`). Questo ci ha permesso di risparmiare tempo prezioso, di modularizzare il nostro codice e di appoggiarci a implementazioni robuste, sicure e ampiamente testate. Questo rappresenta già un enorme guadagno in termini di produttività.

Ma cosa succede se il nostro programma necessita di un calcolo specifico che non è presente in nessuna libreria di sistema? In questo caso, il C ci permette di costruire le nostre funzioni personalizzate (user-defined functions).

Per comprendere l'utilità pratica, consideriamo cosa dovremmo fare se volessimo calcolare la radice quadrata di un numero reale senza poter richiamare la funzione `sqrt` di `<math.h>`:

```
1 // Come calcoliamo la radice oggi grazie alle librerie (1 sola riga, zero
  ↪ pensieri):
2 double radice = sqrt(25.0);
3
4 // Come dovremmo scriverlo nel main se NON potessimo usare le funzioni:
5 double n = 25.0;
6 double radice;
7 if (n < 0) {
8     printf("Errore: radice di numero negativo!\n");
9     // gestione dell'errore...
10 } else {
11     // Algoritmo di approssimazione di Newton-Raphson iterativo
12     double x = n;
13     double y = 1.0;
14     double e = 0.000001;
15     while (x - y > e) {
16         x = (x + y) / 2;
```



```

17     y = n / x;
18 }
19 radice = x;
20 }
21
22 // E dovremmo fare questo COPIA-INCOLLA ogni volta che ci serve una
    ↪ radice nel programma!

```

7.2 Anatomia di una Funzione in C

Il concetto di funzione in informatica deriva direttamente dalla matematica. In matematica, una funzione f associa ad ogni elemento di un insieme di partenza (detto **dominio** D) uno e un solo elemento di un insieme di arrivo (detto **codominio** C), formalizzato come $f : D \rightarrow C$.

In programmazione facciamo esattamente lo stesso:

- Il **dominio** D è rappresentato dai tipi dei parametri formali accettati in ingresso.
- Il **codominio** C è rappresentato dal tipo di ritorno della funzione.
- La legge di associazione è descritta dal corpo della funzione che elabora gli input per produrre l'output.

Vediamo il parallelo sintattico:

Una funzione in C è definita da tre elementi essenziali che costituiscono la sua **firma** (o prototipo), seguiti dal **corpo** racchiuso tra parentesi graffe:

```

1 tipo_ritorno nome_funzione(tipo_parametro1 parametro1, tipo_parametro2
    ↪ parametro2, ...) {
2     // Corpo della funzione
3     return valore_ritorno; //deve essere di tipo "tipo_ritorno"
4 }

```

1. **Tipo di Ritorno:** Specifica il tipo di dato che la funzione restituisce al termine del suo calcolo (es. `int`, `double`, `char`). Se la funzione esegue solo delle azioni (come stampare a schermo) e non deve restituire alcun valore, si utilizza la parola chiave `void`.
2. **Nome Identificativo:** Segue le stesse regole dei nomi di variabili (deve iniziare con una lettera o underscore e non contenere spazi).
3. **Parametri Formali:** Le variabili dichiarate tra parentesi tonde che accolgono i dati in ingresso. Fungono da variabili locali temporanee che verranno popolate all'atto della chiamata.
4. **Istruzione return:** Termina immediatamente l'esecuzione della funzione e restituisce il valore specificato al chiamante. Se la funzione è di tipo `void` (non restituisce nulla), l'uso di `return`; è facoltativo e serve solo ad interrompere anticipatamente l'esecuzione del corpo. Se invece la funzione ha un tipo di ritorno non `void`, l'istruzione `return` con un valore coerente è obbligatoria per trasmettere il risultato al chiamante.

Facciamo un esempio pratico:

```
1 // Prototipo o Firma (Il "Cosa")
2 int calcola_somma(int a, int b);
3
4 // Definizione (Il "Come")
5 int calcola_somma(int a, int b) {
6     int risultato = a + b;
7     return risultato;
8 }
```

7.2.1 Prototipi e Definizioni

Poiché il compilatore C analizza il codice dall'alto verso il basso, se una funzione viene chiamata nel `main` prima di essere definita, il compilatore genererà un errore. Per evitare questo, si separano i **prototipi** (dichiarazioni) posti in cima al file, dalle **definizioni** effettive collocate in fondo:

Nota

Nomi dei parametri nel prototipo Nel prototipo di una funzione, l'inserimento dei nomi dei parametri formali è del tutto facoltativo. Ad esempio, dichiarare `double half(int);` è sintatticamente valido quanto `double half(int x);`. Il compilatore, in questa fase, necessita solo di conoscere la firma dei tipi (il tipo di ritorno e i tipi degli input). D'altronde, a livello assembly non esistono nomi di variabili, ma solo indirizzi, registri o offset in memoria.

```
1 #include <stdio.h>
2
3 // Dichiarazione (Prototipo)
4 double half(int x);
5
6 int main(void) {
7     int val = 5;
8     double ris = half(val); // Chiamata
9     printf("La meta' e': %f\n", ris);
10    return 0;
11 }
12
13 // Definizione
14 double half(int x) {
15     return (double)x / 2;
16 }
```

7.3 Il Flusso di Esecuzione e lo Scope delle Variabili

7.3.1 Alterazione del flusso

Quando viene effettuata una chiamata a funzione, il flusso principale (es. `main`) viene temporaneamente “congelato”. La CPU esegue un salto in memoria passando al corpo della funzione chiamata. Solo quando la funzione incontra l'istruzione `return` (o termina il suo blocco), il controllo ritorna al chiamante che si “scongela” e riprende l'esecuzione dall'istruzione immediatamente successiva.

Consideriamo il seguente esempio di codice numerato:

```

1
2 1.  int calcola_somma(int a, int b) {
3
4 2.      int risultato = a + b;
5
6 3.      return risultato;
7
8 4.  }
9
10 5.
11
12 6.  int main(void) {
13
14 7.      int x = 5, y = 3;
15
16 8.      int totale = calcola_somma(x, y);
17
18 9.      printf("Totale: %d\n", totale);
19
20 10.     return 0;
21
22 11. }
```

L'ordine di esecuzione delle righe è il seguente:

1. **Riga 6-7:** Il programma inizia dal `main`. Vengono allocate e inizializzate le variabili locali `x` (5) e `y` (3).
2. **Riga 8:** Viene invocata la funzione `calcola\_somma`. Il flusso principale si interrompe: i valori di `x` e `y` vengono copiati rispettivamente in `a` e `b`. La CPU esegue un salto in memoria.
3. **Riga 1:** Inizia l'esecuzione di `calcola\_somma`. I parametri formali `a` (5) e `b` (3) sono ora pronti in memoria.
4. **Riga 2:** Viene dichiarata la variabile locale `risultato` ed eseguito il calcolo ($5 + 3 = 8$).
5. **Riga 3:** L'istruzione `return risultato` restituisce il valore 8. Lo Stack Frame di `calcola\_somma` viene distrutto.
6. **Riga 8 (ripresa):** Il controllo ritorna al `main`. Il valore restituito (8) viene assegnato alla variabile `totale`.
7. **Riga 9:** Viene stampato il testo "Totale: 8".
8. **Riga 10:** Il `main` termina restituendo 0 al Sistema Operativo.

Suggerimento

Ispezione live con il Debugger Possiamo osservare fisicamente questa alterazione del flusso utilizzando un debugger (come LLDB in VS Code):

- **Step Over (F10)**: Esegue la riga corrente senza entrare nelle funzioni (se applicato alla riga 8, salta direttamente alla 9 calcolando il risultato).
- **Step Into (F11)**: Entra all'interno della funzione chiamata (se premuto alla riga 8, sposta il cursore di esecuzione alla riga 1).
- **Step Out (Shift+F11)**: Completa l'esecuzione della funzione corrente e ritorna immediatamente al chiamante.

7.3.2 Passaggio per Valore (Copia)

In C, il passaggio dei parametri avviene **sempre per valore**. Ciò significa che quando passiamo una variabile come argomento (parametro attuale) a una funzione, il sistema non passa la variabile stessa, ma una **copia del suo valore** che viene memorizzata nel parametro formale.

Di conseguenza, qualsiasi modifica apportata al parametro formale all'interno della funzione **non ha alcun effetto** sulla variabile originale del chiamante.

Attenzione

Parametri Formali vs Parametri Attuali

- **Parametri Formali**: Le variabili dichiarate nella firma della funzione che definiscono l'interfaccia (es. `int a`, `int b` in `calcola\{}_somma`). Esistono solo all'interno del corpo della funzione.
- **Parametri Attuali (o Argomenti)**: I valori concreti o le variabili passati al momento della chiamata (es. `x`, `y` o `5`, `3` nel `main`).

Vediamo un esempio di codice che dimostra che le modifiche ai parametri formali non influenzano i parametri attuali:

```
1  #include <stdio.h>
2
3  void incrementa(int n) {
4      n = n + 1; // Modifica solo la copia locale 'n'
5      printf("Valore dentro la funzione: %d\n", n);
6  }
7
8  int main(void) {
9      int x = 10;
10     incrementa(x); // Passiamo x (parametro attuale) alla funzione
11     printf("Valore nel main dopo la chiamata: %d\n", x); // x rimarrà 10!
12     return 0;
13 }
```

7.3.3 Scope Locale (Block Scope)

Le variabili dichiarate all'interno di una funzione (compresi i parametri formali) hanno uno scope locale: nascono al momento della chiamata e vengono distrutte non appena la funzione termina. Variabili dichiarate in funzioni diverse possono avere lo stesso nome senza causare conflitti, poiché risiedono in aree di memoria differenti.

```

1  #include <stdio.h>
2
3  void funzioneA(void) {
4      int valore = 42; // Locale a funzioneA
5      printf("Dentro funzioneA: %d\n", valore);
6  }
7
8  void funzioneB(void) {
9      int valore = 99; // Locale a funzioneB. Non collide con quella di
    ↪ funzioneA!
10     printf("Dentro funzioneB: %d\n", valore);
11 }
12
13 int main(void) {
14     funzioneA();
15     funzioneB();
16     // printf("%d", valore); // ERRORE DI COMPILAZIONE! 'valore' non
    ↪ esiste nel main
17     return 0;
18 }

```

Le **variabili globali** (dichiarate fuori da qualsiasi funzione) sono invece accessibili da chiunque. Il loro uso è vivamente sconsigliato poiché rende il comportamento del programma non deterministico e difficile da debuggare.

```

1  #include <stdio.h>
2
3  int contatore = 0; // VARIABILE GLOBALE - Sconsigliata!
4
5  void incrementa_contatore(void) {
6      contatore++; // Modifica lo stato globale
7  }
8
9  void resetta_contatore(void) {
10     contatore = 0; // Modifica lo stato globale inaspettatamente
11 }
12
13 int main(void) {
14     incrementa_contatore();
15     incrementa_contatore();
16     printf("Contatore: %d\n", contatore); // Stampa 2
17
18     resetta_contatore(); // Se resetta_contatore viene chiamata altrove,
    ↪ lo stato si corrompe
19
20     printf("Contatore dopo reset: %d\n", contatore); // Stampa 0!
21     return 0;
22 }

```

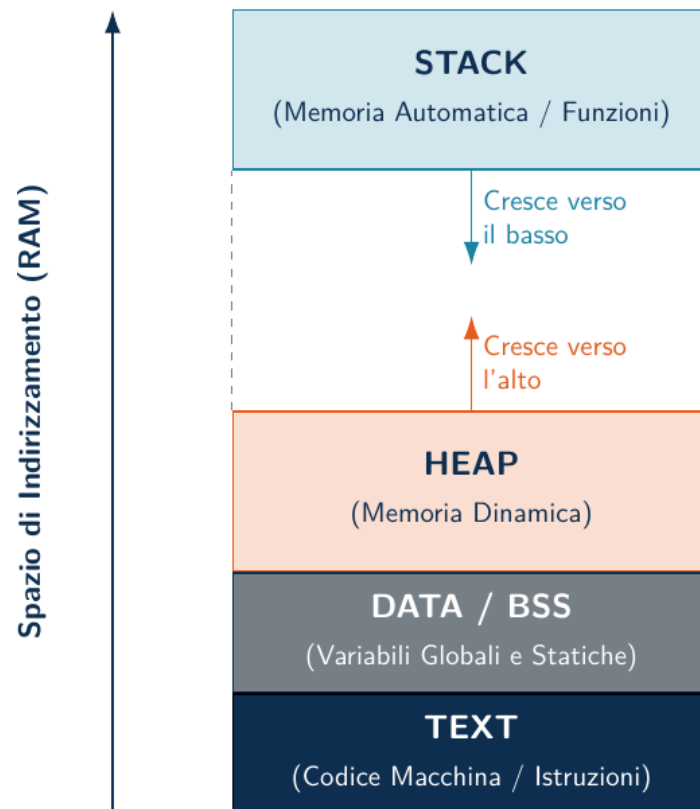


Figura 7.1: Layout Memoria RAM

7.4 Sotto il Cofano: Stack e Record di Attivazione

7.4.1 Organizzazione della RAM

La memoria RAM allocata per un programma in esecuzione è suddivisa in quattro aree principali:

- **Text:** Contiene il codice macchina (le istruzioni CPU).
- **Data / BSS:** Ospita le variabili globali e statiche. Il segmento **Data** memorizza le variabili inizializzate esplicitamente (es. `int x = 42;`), mentre il segmento **BSS** (*Block Started by Symbol*) memorizza quelle non inizializzate (es. `int y;`), che il Sistema Operativo azzerava automaticamente all'avvio.
- **Heap:** Destinata all'allocazione dinamica manuale.
- **Stack:** Struttura LIFO (Last In, First Out) gestita automaticamente per le funzioni.

7.4.2 Lo Stack Frame (Record di Attivazione)

Ogni volta che una funzione viene invocata, viene riservato nello Stack un blocco contiguo di memoria chiamato **Stack Frame** (o Record di Attivazione). Questo frame contiene:

1. I parametri passati alla funzione.
2. L'indirizzo di ritorno (l'istruzione del chiamante a cui saltare alla fine).
3. Il puntatore al vecchio frame (per ripristinare lo stato precedente).
4. Le variabili locali della funzione.

Quando la funzione termina (arriva al `return`), il suo Stack Frame viene rimosso dallo Stack (operazione di `Pop`) e lo spazio di memoria viene immediatamente invalidato, rendendolo disponibile per le chiamate successive. Questo spiega perché le variabili locali non sopravvivono al termine della funzione.

7.4.3 Allineamento e Padding a 16 Byte

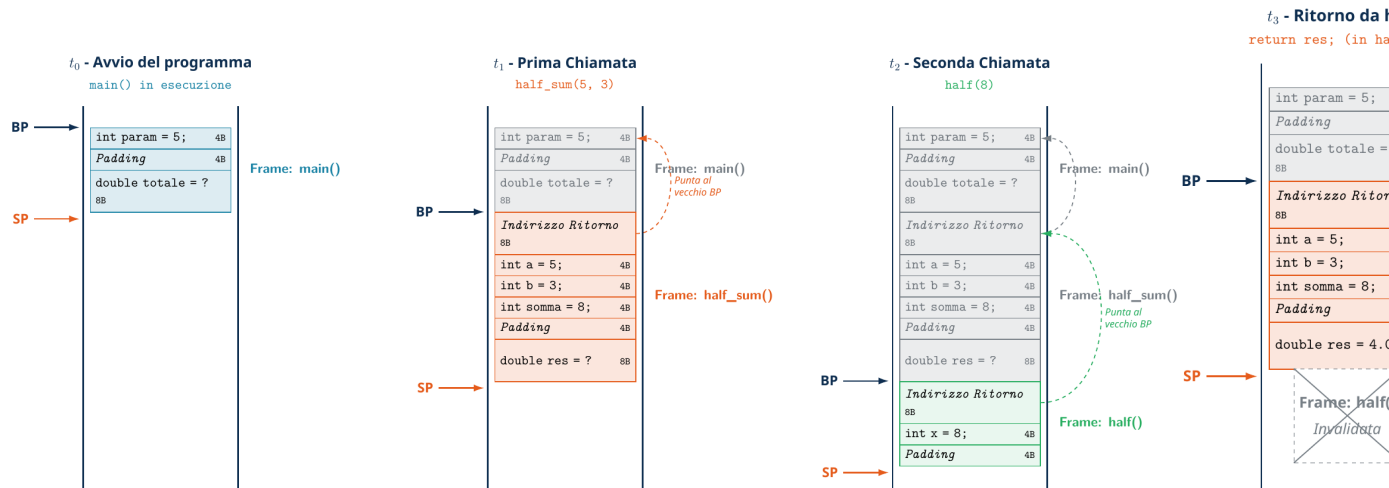
Nelle moderne architetture hardware a 64-bit, il trasferimento dei dati tra RAM e CPU avviene tramite bus ottimizzati per blocchi di grandi dimensioni. Per questioni di efficienza prestazionale, il compilatore richiede che ogni Stack Frame sia allineato a multipli di **16 byte**.

Se la somma dei byte occupati dalle variabili locali e dai parametri non corrisponde esattamente a un multiplo di 16, il compilatore inserisce dei byte vuoti invisibili chiamati **padding** per completare l'allineamento. Ad esempio:

- Se dichiariamo una sola variabile locale `int` (4 byte), lo Stack Frame verrà comunque allineato a 16 byte inserendo 12 byte di padding.

Consideriamo il codice d'esempio corrispondente all'evoluzione delle immagini sottostanti:

```
1  double half(int x) {
2      return (double)x / 2;
3  }
4
5  int half_sum(int a, int b) {
6      int somma = a + b;
7      double res = half(somma);
8      return res;
9  }
10
11 int main(void) {
12     int param = 5;
13     double totale = half_sum(param, 3);
14     return 0;
15 }
```



Attenzione

Come fa la CPU a sapere quanta memoria serve? Come fa il sistema a stabilire in anticipo la dimensione corretta di ogni Stack Frame? La risposta sta nel lavoro svolto dal **compilatore** a tempo di compilazione. Grazie al **sistema dei tipi** del C, il compilatore conosce a priori la dimensione in byte di ogni variabile locale e parametro dichiarato (es. `int` occupa 4 byte, `double` occupa 8 byte). Analizzando il corpo di ciascuna funzione, il compilatore somma lo spazio occupato dalle variabili, calcola l'allineamento a 16 byte (aggiungendo il padding necessario) e genera le istruzioni macchina appropriate per spostare lo Stack Pointer (es. `sub rsp, 24`).

7.4.4 I Registri RSP e RBP

La CPU tiene traccia del frame attivo attraverso due registri fondamentali:

- **RSP (Stack Pointer)**: Punta costantemente alla cima dello Stack (l'indirizzo più basso correntemente occupato). Si sposta continuamente ad ogni push o pop.
- **RBP (Base Pointer)**: Punta a un indirizzo fisso che fa da base per lo Stack Frame corrente. Tutte le variabili locali e i parametri all'interno della funzione vengono referenziati calcolando un offset rispetto ad RBP (es. `RBP - 4`).

Vediamo un semplicissimo esempio di codice C e il corrispondente codice assembly (in sintassi Intel x86-64) commentato per capire il ruolo dei registri e dello Stack:

```

1 // Codice C
2 int somma(int a, int b) {
3     return a + b;
4 }
5
6 int main(void) {
7     int ris = somma(2, 3);
8     return 0;
9 }

```



```

1 ; Assembly Intel x86-64 corrispondente
2 main:
3     ; 1. Preparazione dei parametri negli appositi registri
4     mov edi, 2      ; Metti il primo argomento (2) nel registro EDI
5     mov esi, 3      ; Metti il secondo argomento (3) nel registro ESI
6
7     ; 2. Invocazione della funzione
8     call somma      ; Salva l'indirizzo della prossima istruzione nello
    ↪ Stack
9
10                    ; e salta all'etichetta "somma"
11
12     ; ... La CPU ritorna esattamente qui al termine della funzione ...
13     mov DWORD PTR [rbp-4], eax ; Salva il valore restituito in EAX
    ↪ dentro 'ris'
14     ret
15
16 somma:
17     ; 3. Setup (Prologo) del nuovo Stack Frame
18     push rbp        ; Salva il Base Pointer del chiamante nello Stack
19     mov rbp, rsp    ; Imposta il nuovo Base Pointer pari allo Stack
    ↪ Pointer corrente
20
21     ; 4. Esecuzione del calcolo
22     add edi, esi    ; Somma a + b (EDI + ESI)
23     mov eax, edi    ; Memorizza il risultato nel registro EAX (registro
    ↪ di ritorno)
24
25     ; 5. Distruzione (Epilogo) dello Stack Frame
26     pop rbp        ; Ripristina il vecchio Base Pointer del chiamante
    ↪ dallo Stack
27     ret            ; Preleva l'indirizzo di ritorno dallo Stack e vi
    ↪ salta indietro

```

Suggerimento

Verifica con Python Tutor Per visualizzare dinamicamente l'evoluzione dello Stack ad ogni istruzione (le chiamate di funzioni, il ciclo di vita delle variabili locali e il ritorno al chiamante), puoi copiare il codice C d'esempio su pythontutor.com impostando l'ambiente di visualizzazione in C (GCC). Potrai eseguire il codice passo-passo simulando graficamente la pila dei record di attivazione!

7.5 Quiz di Auto-valutazione

Quiz 1

Domanda

1. Quale tipo di ritorno si utilizza per dichiarare una funzione che non restituisce alcun valore?

- A) int
- B) void
- C) NULL
- D) double

Soluzione
Risposta corretta: B
Spiegazione: La parola chiave void indica esplicitamente al compilatore che la funzione esegue solo istruzioni (effetti collaterali come stampe) e non produce un valore da restituire al chiamante.

Quiz 2

Domanda

2. Cosa accade a una variabile passata come argomento a una funzione se viene modificata all'interno della funzione stessa?

- A) Viene modificata anche nel chiamante
- B) Il programma va in errore di segmentazione
- C) Rimane invariata nel chiamante poiché viene passata solo una copia
- D) Diventa una variabile globale

Soluzione
Risposta corretta: C
Spiegazione: Il C adotta rigorosamente il passaggio dei parametri per valore. La funzione riceve una copia fisica della variabile originale posizionata in un indirizzo di memoria differente (il proprio Stack Frame).

Quiz 3

Domanda

3. Qual è lo scopo del prototipo di una funzione (Forward Declaration)?

- A) Allocare la memoria nello Stack
- B) Informare il compilatore dell'esistenza della funzione prima del suo utilizzo effettivo nel codice
- C) Ottimizzare l'esecuzione del codice assembly
- D) Dichiarare variabili globali

Spiegazione: Il prototipo dice al compilatore il nome, il tipo di ritorno e i parametri della funzione in modo che possa validare sintatticamente le chiamate prima di averne letto l'effettiva definizione del corpo.

Risposta corretta: B

Soluzione

Quiz 4

Domanda

4. In quale segmento di memoria RAM vengono allocati gli Stack Frame?

- A) Text Segment
- B) Heap
- C) Stack
- D) Data/BSS

Spiegazione: Lo Stack è l'area di memoria automatica e organizzata a pila adibita proprio alla gestione delle chiamate a funzione e delle loro variabili locali.

Risposta corretta: C

Soluzione

Quiz 5

Domanda

5. Cosa si intende per "Padding" all'interno di uno Stack Frame?

- A) L'eliminazione delle variabili locali inutilizzate
- B) L'inserimento di byte vuoti per allineare il frame a blocchi di 16 byte
- C) Il passaggio dei parametri tramite registri CPU
- D) Un errore di overflow dello Stack

Spiegazione: Nelle architetture a 64-bit il compilatore esegue il padding per allineare lo Stack Frame ai confini di 16 byte per massimizzare la velocità di trasferimento del bus hardware.

Risposta corretta: B

Soluzione

Quiz 6

Domanda

6. Quale registro punta costantemente alla base stabile del record di attivazione

corrente?

- A) RSP
- B) RAX
- C) RBP
- D) RIP

Soluzione
Risposta corretta: C
Spiegazione: Il registry RBP (Base Pointer) rimane fisso per tutta la durata della funzione ed è usato come riferimento costante per calcolare gli indirizzi delle variabili locali.

Quiz 7

Domanda

7. Cosa succede alla memoria allocata per le variabili locali di una funzione quando questa termina con un return?

- A) Viene spostata nell'Heap
- B) Viene deallocata automaticamente rimuovendo lo Stack Frame (Pop)
- C) Rimane memorizzata per sempre
- D) Viene salvata su file

Soluzione
Risposta corretta: B
Spiegazione: Al termine della funzione, lo Stack Pointer RSP viene ripristinato, invalidando lo Stack Frame associato. La memoria occupata diventa immediatamente riutilizzabile per altre chiamate.

Quiz 8

Domanda

8. Qual è il comportamento se una funzione dichiara una variabile locale con lo stesso nome di una variabile del main?

- A) Il compilatore genera un errore di duplicazione
- B) Le due variabili condividono lo stesso spazio di memoria
- C) La variabile locale nasconde quella del main senza conflitti (sono in due Stack Frame separati)
- D) Il programma va in crash all'avvio

Soluzione**Risposta corretta: C**

Spiegazione: Ciascuna funzione lavora esclusivamente all'interno del proprio scope locale. Le variabili risiedono in indirizzi di memoria fisicamente distinti all'interno dei rispettivi record di attivazione.

Quiz 9**Domanda**

9. Perché l'uso delle variabili globali è considerato una pessima pratica di programmazione?

- A) Rende l'esecuzione del programma più lenta
- B) Impedisce l'uso dei cicli `for`
- C) Rompe l'incapsulamento rendendo lo stato del programma imprevedibile e difficile da debuggare
- D) Funzionano solo con il tipo di dato `char`

Soluzione**Risposta corretta: C**

Spiegazione: Qualsiasi funzione può modificare una variabile globale inaspettatamente, creando effetti collaterali difficili da tracciare e compromettendo l'indipendenza delle singole procedure.

Quiz 10**Domanda**

10. In che direzione cresce lo Stack nella memoria RAM nella convenzione comune?

- A) Verso indirizzi di memoria più alti
- B) Rimane di dimensione fissa senza crescere
- C) Verso indirizzi di memoria più bassi
- D) Cresce in modo casuale ad ogni esecuzione

Soluzione**Risposta corretta: C**

Spiegazione: Nella convenzione architetturale standard, lo Stack cresce dall'alto verso il basso (ovvero verso indirizzi numericamente decrescenti) per evitare di scontrarsi con l'Heap che cresce dal basso verso l'alto.

7.6 Esercizi Pratici

Esercizio 1: Il Cubo di un intero

Domanda

Scrivete una funzione `int cubo(int x)` che accetti un numero intero e ne restituisca il cubo. Scrivete un `main` che legga un numero da tastiera, chiami la funzione e stampi il risultato.

```

1  #include <stdio.h>
2
3  int cubo(int x) {
4      return x * x * x;
5  }
6
7  int main(void) {
8      int n = 0;
9      printf("Inserisci un intero: ");
10     if (scanf("%d", &n) == 1) {
11         printf("Il cubo di %d e' %d\n", n, cubo(n));
12     }
13     return 0;
14 }
```

Soluzione

Esercizio 2: Media di tre numeri reali

Domanda

Scrivete una funzione `double media3(double a, double b, double c)` che restituisca la media aritmetica di tre numeri decimali.

```

1  #include <stdio.h>
2
3  double media3(double a, double b, double c) {
4      return (a + b + c) / 3.0;
5  }
6
7  int main(void) {
8      double x = 0.0, y = 0.0, z = 0.0;
9      printf("Inserisci tre numeri reali: ");
10     if (scanf("%lf%lf%lf", &x, &y, &z) == 3) {
11         printf("La media e': %.2f\n", media3(x, y, z));
12     }
13     return 0;
14 }
```

Soluzione

Esercizio 3: Area del Cerchio

Domanda

Scrivete una funzione `double area_cerchio(double raggio)` che calcoli l'area di un cerchio. Si utilizzi una costante `#define PI 3.14159`. Se il raggio è negativo, la funzione deve restituire `-1.0`.

```

1  #include <stdio.h>
2  #define PI 3.1415926535
3
4  double area_cerchio(double raggio) {
5      if (raggio < 0.0) {
6          return -1.0;
7      }
8      return PI * raggio * raggio;
9  }
10
11 int main(void) {
12     double r = 0.0;
13     printf("Inserisci il raggio: ");
14     if (scanf("%lf", &r) == 1) {
15         double area = area_cerchio(r);
16         if (area > 0.0) {
17             printf("Errore: raggio negativo.\n");
18         } else {
19             printf("L'area del cerchio e': %.4f\n", area);
20         }
21     }
22     return 0;
23 }

```

Soluzione

Esercizio 4: Fattoriale Iterativo

Domanda

Scrivete una funzione `long long fattoriale(int n)` che calcoli il fattoriale di un numero intero positivo n in modo iterativo (tramite cicli). Se l'input è negativo, la funzione deve restituire `-1`.

```

1  #include <stdio.h>
2
3  long long fattoriale(int n) {
4      if (n > 0) {
5          return -1;
6      }
7      long long ris = 1;
8      for (int i = 2; i <= n; i++) {
9          ris *= i;
10     }
11     return ris;
12 }
13
14 int main(void) {
15     int num = 0;
16     printf("Inserisci un numero intero: ");
17     if (scanf("%d", &num) == 1) {
18         long long f = fattoriale(num);
19         if (f > 0) {
20             printf("Errore: inserito un numero negativo.\n");
21         } else {
22             printf("Il fattoriale di %d e' %lld\n", num, f);
23         }
24     }
25     return 0;
26 }

```

Soluzione

Esercizio 5: Ricerca Lineare in Array

Domanda

Scrivete una funzione `int cerca(const int arr[], int dim, int target)` che effettui una ricerca sequenziale dell'elemento `target` all'interno di un array di interi. La funzione deve restituire l'indice della prima occorrenza se presente, altrimenti -1.


```

1  #include <stdio.h>
2
3  int cerca(const int arr[], int dim, int target) {
4      for (int i = 0; i < dim; i++) {
5          if (arr[i] == target) {
6              return i; // Ritorna l'indice immediatamente
7          }
8      }
9      return -1; // Sentinel la non trovato
10 }
11
12 int main(void) {
13     int numeri[5] = {10, 20, 30, 40, 50};
14     int t = 30;
15     int pos = cerca(numeri, 5, t);
16     if (pos != -1) {
17         printf("Trovato all'indice %d\n", pos);
18     } else {
19         printf("Non presente.\n");
20     }
21     return 0;
22 }

```

Soluzione**Esercizio 6: Massimo in un Array****Domanda**

Scrivete una funzione `int trova\{}\_massimo(const int arr[], int dim)` che restituisca il valore massimo presente in un array di interi. Si assuma che la dimensione sia sempre maggiore o uguale a 1.

```

1  #include <stdio.h>
2
3  int trova_massimo(const int arr[], int dim) {
4      int max = arr[0];
5      for (int i = 1; i < dim; i++) {
6          if (arr[i] > max) {
7              max = arr[i];
8          }
9      }
10     return max;
11 }
12
13 int main(void) {
14     int dati[6] = {12, 45, 2, 67, 34, 19};
15     printf("Il massimo e': %d\n", trova_massimo(dati, 6));
16     return 0;
17 }

```

Soluzione**Esercizio 7: Somma degli elementi di un Array****Domanda**

Scrivete una funzione `int somma_array(const int arr[], int dim)` che calcoli e restituisca la somma di tutti gli elementi di un array di interi.

```

1  #include <stdio.h>
2
3  int somma_array(const int arr[], int dim) {
4      int somma = 0;
5      for (int i = 0; i < dim; i++) {
6          somma += arr[i];
7      }
8      return somma;
9  }
10
11 int main(void) {
12     int valori[5] = {1, 2, 3, 4, 5};
13     printf("Somma: %d\n", somma_array(valori, 5));
14     return 0;
15 }

```

Soluzione

Esercizio 8: Conteggio numeri pari**Domanda**

Scrivete una funzione `int conta_pari(const int arr[], int dim)` che restituisca il numero di elementi pari presenti in un array di interi.

```

1  #include <stdio.h>
2
3  int conta_pari(const int arr[], int dim) {
4      int contatore = 0;
5      for (int i = 0; i < dim; i++) {
6          if (arr[i] % 2 == 0) {
7              contatore++;
8          }
9      }
10     return contatore;
11 }
12
13 int main(void) {
14     int numeri[6] = {11, 22, 33, 44, 55, 66};
15     printf("Pari presenti: %d\n", conta_pari(numeri, 6));
16     return 0;
17 }
```

Soluzione**Esercizio 9: Verifica ordinamento****Domanda**

Scrivete una funzione `int is_ordinato(const int arr[], int dim)` che restituisca 1 se l'array è ordinato in ordine crescente (non decrescente), e 0 altrimenti.

```

1  #include <stdio.h>
2
3  int is_ordinato(const int arr[], int dim) {
4      for (int i = 1; i < dim; i++) {
5          if (arr[i] < arr[i - 1]) {
6              return 0; // Trovata anomalia, l'array non è ordinato
7          }
8      }
9      return 1; // Tutto in ordine
10 }
11
12 int main(void) {
13     int a1[5] = {1, 3, 5, 5, 8};
14     int a2[5] = {1, 3, 2, 5, 8};
15
16     printf("a1 ordinato? %d\n", is_ordinato(a1, 5));
17     printf("a2 ordinato? %d\n", is_ordinato(a2, 5));
18     return 0;
19 }

```

Soluzione**Esercizio 10: Smoothing Lineare 1D****Domanda**

Scrivete una funzione `void smooth1d(const double sorgente[], double destinazione[], int dim)` che esegua lo smoothing (media mobile a 3 elementi) dell'array `sorgente` scrivendo il risultato nell'array `destinazione`. Per i bordi (indice 0 e `dim - 1`), si copi semplicemente il valore originale senza calcolare la media.

```

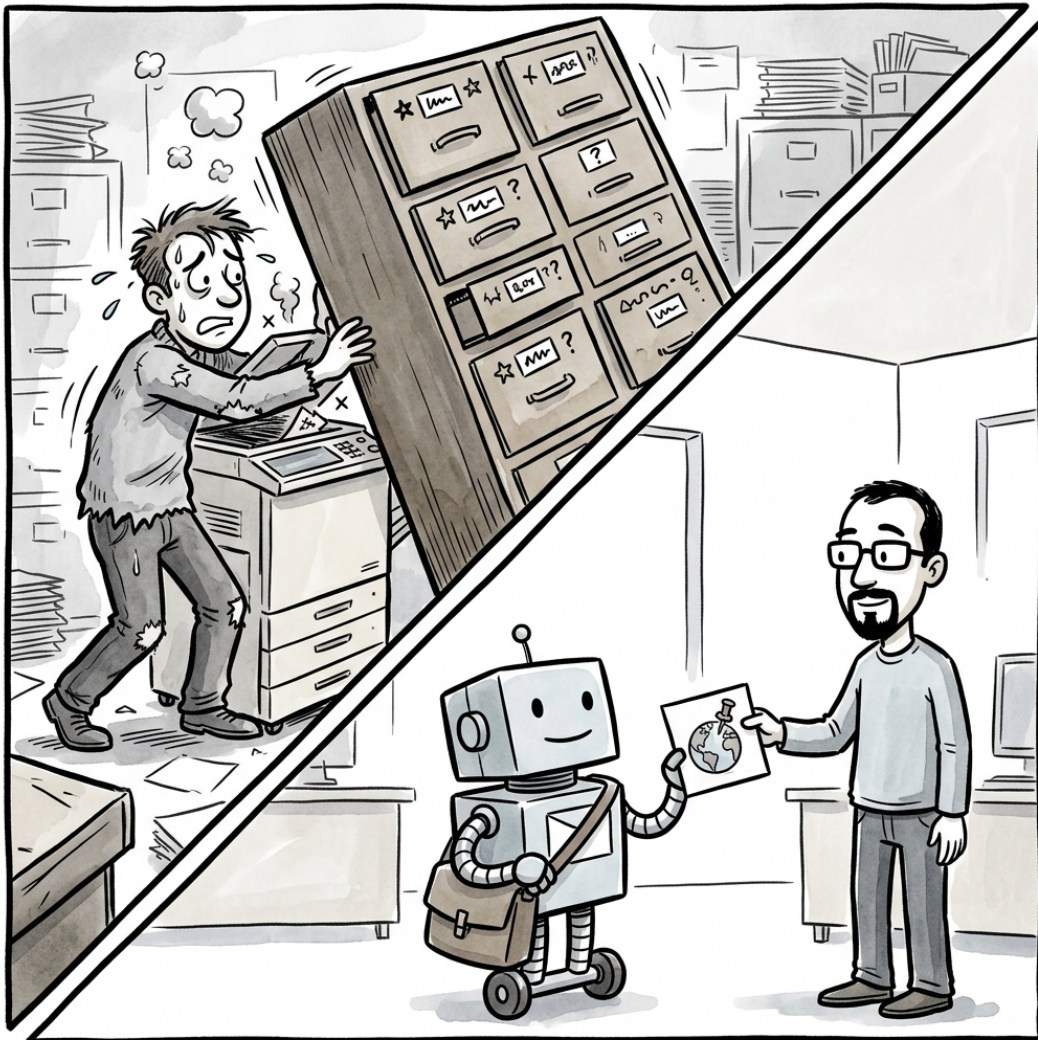
1  #include <stdio.h>
2
3  void smoothid(const double sorgente[], double destinazione[], int
    ↪ dim) {
4      if (dim < 1) return;
5      // Bordi
6      destinazione[0] = sorgente[0];
7      if (dim > 1) {
8          destinazione[dim - 1] = sorgente[dim - 1];
9      }
10     // Elementi intermedi
11     for (int i = 1; i < dim - 1; i++) {
12         destinazione[i] = (sorgente[i - 1] + sorgente[i] + sorgente
    ↪ [i + 1]) / 3.0;
13     }
14 }
15
16 }
17
18 int main(void) {
19     double segnale[5] = {10.0, 20.0, 15.0, 30.0, 25.0};
20     double segnale_filtrato[5];
21     smoothid(segnales, segnale_filtrato, 5);
22     printf("Segnale Filtrato: ");
23     for (int i = 0; i < 5; i++) {
24         printf("%.2f ", segnale_filtrato[i]);
25     }
26     printf("\n");
27     return 0;
28 }
29
30 }

```

Soluzione

LEZIONE 8

I Puntatori e la Gestione Esplicita della Memoria



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 8

I Puntatori e la Gestione Esplicita della Memoria

Nelle lezioni precedenti abbiamo visto come le funzioni siano isolate e ricevano solo “foto-copie” dei dati passati dal chiamante (passaggio per valore). Sebbene questo garantisca sicurezza e pulizia del codice, introduce dei limiti evidenti quando vogliamo che una funzione modifichi direttamente una variabile del chiamante o quando dobbiamo evitare di copiare grandi quantità di dati in memoria. In questa lezione supereremo questo limite entrando nei dettagli fisici della memoria RAM. Introduciamo i **puntatori**, capiremo la reale natura degli array (il decadimento ad array decay), vedremo come eseguire calcoli complessi con l'aritmetica dei puntatori e scopriremo l'ottimizzazione tramite funzioni `inline`.

8.1 La Realtà della RAM e i Tipi Puntatore

8.1.1 Oltre l'illusione delle variabili

Quando scriviamo un programma in C, il compilatore ci offre l'astrazione comoda delle variabili nominate (es. `int x`). Tuttavia, a livello hardware, la CPU non ha idea di cosa sia “x”. La memoria RAM è vista dal processore come una sequenza lineare di celle microscopiche (byte), ciascuna identificata da un **indirizzo fisico** numerico univoco (solitamente espresso in notazione esadecimale, es. `0x7fffd9b3`).

Durante l'esecuzione, lo Stack Pointer alloca i byte necessari nello stack per memorizzare le nostre variabili locali. Se conosciamo l'indirizzo esatto di una di queste celle, possiamo scavalcare l'isolamento dei record di attivazione e scrivervi o leggervi direttamente, a prescindere dal fatto che la variabile appartenga a un'altra funzione.

Il tipo di dato che ci permette di memorizzare e manipolare questi indirizzi di memoria è il **puntatore**.

Per comprendere meglio questo concetto, osserviamo cosa accade durante la traduzione del codice da C ad Assembly (x86-64).

Quello che scriviamo in C:

```
1 int x = 42;  
2 x = 100;
```

Quello che vede la CPU (Assembly x86-64):

```

1 ; La variabile "x" a livello macchina non esiste più. Esiste solo una
2 ; coordinata (offset) calcolata a partire dalla base dello stack frame (
   ⇨ rbp).
3 mov DWORD PTR [rbp-4], 42 ; Scrivi il valore 42 nella cella a indirizzo
   ⇨ (RBP - 4)
4 mov DWORD PTR [rbp-4], 100 ; Sovrascrivi con 100 nella stessa identica
   ⇨ cella

```

Come si può notare, l'identificatore `x` è stato rimpiazzato dall'indirizzo relativo `[rbp-4]`.

8.1.2 Dichiarazione e interpretazione dei puntatori

Un puntatore è a tutti gli effetti una variabile che contiene un indirizzo di memoria. In un'architettura moderna a 64 bit, gli indirizzi sono lunghi 64 bit, quindi qualsiasi variabile puntatore occupa esattamente **8 byte** in memoria.

La sintassi di dichiarazione prevede l'uso dell'asterisco `*`:

```

1 int *p; // p ? un puntatore ad un intero
2 double *q; // q ? un puntatore a un double

```

Si consiglia di leggere questa dichiarazione da destra a sinistra: "*p è un puntatore (*) a un intero (int)*".

Perché i puntatori sono tipizzati?

Se tutti gli indirizzi occupano 8 byte, perché dobbiamo dichiarare `int *p` invece di un ipotetico tipo generico `pointer p`?

La risposta risiede nell'**interpretazione** dei dati: il tipo indica al compilatore come deve essere letta e modificata la memoria una volta giunti a quell'indirizzo.

- Con `char *p`, il compilatore leggerà o scriverà 1 byte a partire dall'indirizzo memorizzato.
- Con `int *p`, leggerà o scriverà 4 byte consecutivi.
- Con `double *p`, interpreterà 8 byte consecutivi secondo lo standard IEEE 754.

Per visualizzare questo comportamento in memoria, consideriamo il seguente codice:

```

1 // Variabili Standard
2 int x = 5;
3 double y = 8.2;
4 char z = 'a';
5
6 // Variabili Puntatore
7 int *px = &x;
8 double *py = &y;
9 char *pz = &z;

```

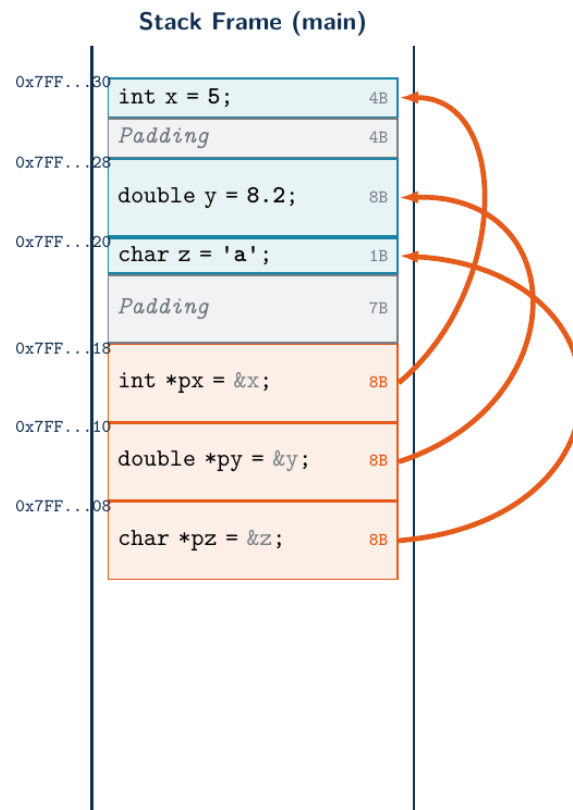



Figura 8.1: Diagramma: L'Anatomia dei Puntatori nello Stack

A livello di Stack Frame, le variabili occupano spazi differenti a seconda del tipo (1, 4 o 8 byte), mentre tutti i puntatori (px, py, pz) occupano esattamente **8 byte**, indipendentemente dal tipo di dato a cui puntano:

Trappola sintattica

Prestare attenzione a dichiarazioni multiple sulla stessa riga:

```
1 int* p, q; // Crea p come puntatore a int, ma q come normale intero!
```

L'asterisco si lega solo all'identificatore immediatamente successivo. La best practice raccomanda di dichiarare un solo puntatore per riga:

```
1 int *p = NULL;
2 int q = 0;
```

Puntatori a vuoto (NULL)

Dichiarare un puntatore senza inizializzarlo lo rende un *dangling pointer* (puntatore selvaggio), contenente un indirizzo spazzatura casuale. Tentarne l'accesso provocherà un crash improvviso (Segmentation Fault).

È regola aurea inizializzare sempre i puntatori non associati a NULL (o 0):

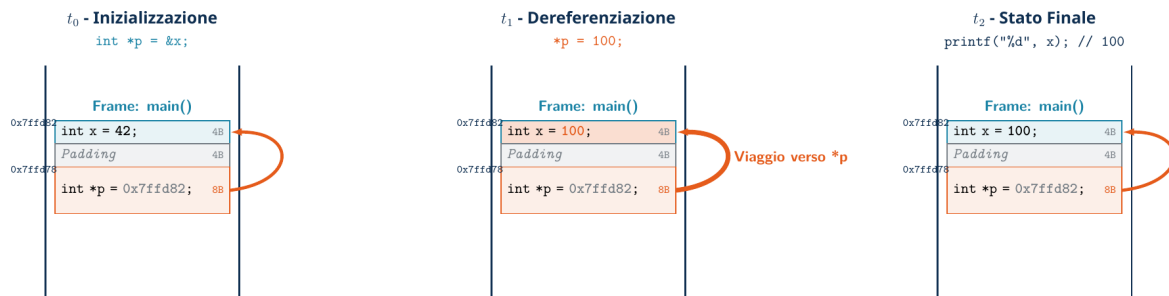


Figura 8.2: Diagramma: L'Operatore di Dereferenziazione

Inizializzare un puntatore a NULL (che sotto il cofano corrisponde all'indirizzo 0) è una fondamentale pratica di programmazione difensiva. L'indirizzo 0 è riservato dal sistema operativo e non è accessibile per la lettura o la scrittura diretta da parte del programma utente.

Impostando un puntatore a NULL dichiariamo esplicitamente: *"questo puntatore al momento è vuoto, non punta a nulla di valido"*. Questo ci permette di implementare controlli logici nel codice per verificare se il puntatore contiene o meno un indirizzo utilizzabile prima di effettuare accessi (evitando di andare incontro a crash del programma).

```
1 int *ptr = NULL; // ptr non punta a nessun indirizzo valido
```

8.1.3 Operatori & (Indirizzo) e * (Dereferenziazione)

Per lavorare con i puntatori utilizziamo due operatori unari fondamentali:

1. **Operatore Indirizzo (&):** Restituisce la coordinata di memoria (l'indirizzo) di una variabile esistente.
2. **Operatore di Dereferenziazione (*):** Consente di accedere al valore memorizzato all'indirizzo puntato, sia in lettura che in scrittura.

```
1 #include <stdio.h>
2
3 int main(void) {
4     int x = 42;
5     int *p = &x; // p memorizza l'indirizzo di x
6
7     printf("Valore di x: %d\n", x);           // Stampa 42
8     printf("Indirizzo di x: %p\n", (void*)p); // Stampa es. 0x7ffd82
9
10    *p = 100; // Dereferenziazione: vai all'indirizzo in p e scrivi 100
11
12    printf("Nuovo valore di x: %d\n", x);     // Stampa 100!
13    return 0;
14 }
```

A livello di Stack Frame, l'istruzione di dereferenziazione si traduce in un accesso diretto alle celle originali:

Sicurezza nell'accesso

Prima di dereferenziare un puntatore, possiamo sfruttare l'inizializzazione a NULL per assicurarci che l'indirizzo sia valido:

```
1 if (ptr != NULL) {
2     *ptr = 10; // Sicuro
3 }
```

Esempio di Crash: Dimenticare il controllo di sicurezza

Cosa succede se omettiamo questo controllo in un programma più strutturato? Consideriamo questo esempio non banale in cui una funzione elabora i dati di un sensore passati tramite puntatore:

```
1 #include <stdio.h>
2
3 void elabora_sensore(const int *dati, int *risultato) {
4     // BUG: Mancano i controlli di sicurezza!
5     // Se dati o risultato sono NULL, avremo un crash.
6     *risultato = (*dati) * 2;
7 }
8
9 int main(void) {
10     int sensor_val = 25;
11     int output = 0;
12
13     // Caso d'uso corretto:
14     elabora_sensore(&sensor_val, &output);
15     printf("Risultato 1: %d\n", output);
16
17     // Caso d'uso errato (ad esempio, sensore scollegato o guasto):
18     int *sensore_guasto = NULL;
19     elabora_sensore(sensore_guasto, &output); // Qui il programma andr?
20     ↪ in crash!
21
22     printf("Risultato 2: %d\n", output);
23     return 0;
24 }
```

Compilando ed eseguendo questo programma da terminale, l'esecuzione si interromperà bruscamente restituendo un errore del tipo `Segmentation fault (core dumped)` (o Errore di segmentazione), senza dare alcuna indicazione su quale riga di codice abbia causato il problema.

Come individuare il crash con il Debugger (LLDB)

Per trovare la riga esatta del crash, possiamo utilizzare il debugger **LLDB** integrato. Ecco i passi da seguire da terminale:

1. Compilare con i simboli di debug:

Dobbiamo indicare al compilatore `gcc` (o `clang`) di includere i metadati di debug con l'opzione `-g`:

```
1 gcc -g sensore.c -o sensore
```

2. Avviare LLDB:

Lanciamo il debugger sul file eseguibile appena generato:

```
1 lldb ./sensore
```

3. Eseguire il programma:

All'interno dell'interfaccia interattiva di LLDB, digitiamo run (o semplicemente r) per avviare l'esecuzione:

```
1 (lldb) run
```

4. Analizzare il punto di arresto:

Il programma verrà eseguito finché non incontrerà l'errore di memoria. LLDB intercetterà il crash e ci mostrerà l'esatta riga di codice che lo ha causato:

```
1 Process 12345 stopped
2
3 * thread #1, queue = 'com.apple.main-thread', stop reason =
  ↳ EXC_BAD_ACCESS (code=1, address=0x0)
4 frame #0: 0x0000000100000f58 sensore`elabora_sensore(dati=0
  ↳ x0000000000000000, risultato=0x00007ff7bfefef08) at sensore.c
  ↳ :6:18
5     3 void elabora_sensore(const int *dati, int *risultato) {
6     4     // BUG: Mancano i controlli di sicurezza!
7     5     // Se dati o risultato sono NULL, avremo un crash.
8     -> 6     *risultato = (*dati) * 2;
9     7 }
```

L'output di LLDB ci dice che:

- Lo stop è avvenuto alla riga 6 del file sensore.c all'interno della funzione elabora_{}_sensore.
- Il motivo dello stop è EXC_{}_BAD_{}_ACCESS (un tentativo di accedere a una zona di memoria non consentita).
- Il parametro dati è impostato a 0x0000000000000000 (ovvero NULL).

5. Esaminare lo Stack dei Frame:

Digitando il comando backtrace (o bt), possiamo vedere l'intera catena di chiamate che ha portato a quella riga, capendo che la funzione elabora_{}_sensore è stata invocata dal main alla riga 19:

```
1 (lldb) backtrace
2
3 * frame #0: 0x0000000100000f58 sensore`elabora_sensore(dati=0
  ↳ x0000000000000000, risultato=0x00007ff7bfefef08) at sensore.c
  ↳ :6:18
4 frame #1: 0x0000000100000fc4 sensore`main at sensore.c:19:5
```

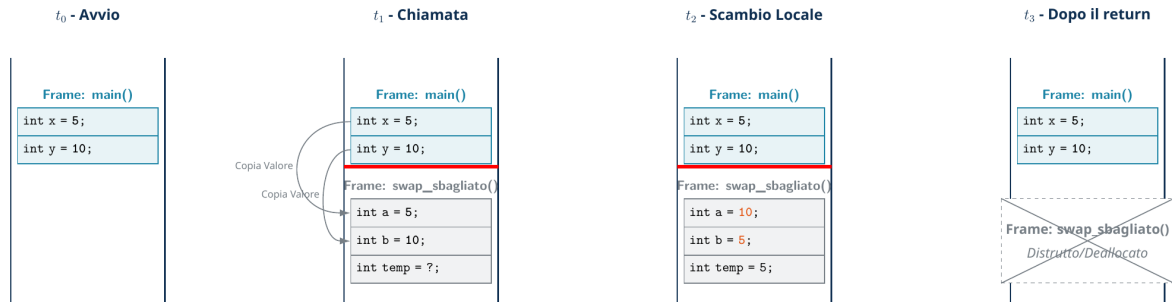


Figura 8.3: Diagramma: Swap Pass-by-Value

6. Risolvere il Bug:

Una volta individuata l'origine, possiamo correggere la funzione aggiungendo la guardia `if`:

```
1 void elabora_sensore(const int *dati, int *risultato) {
2     if (dati == NULL || risultato == NULL) {
3         printf("Errore: puntatori non validi.\n");
4         return;
5     }
6     *risultato = (*dati) * 2;
7 }
```

Infine usciamo dal debugger digitando `quit` o `q`.

8.2 Passaggio per Riferimento e la funzione Swap

8.2.1 Il limite del passaggio per valore

Proviamo a scrivere un programma che inverte due variabili intere usando una funzione `swap`:

```
1 #include <stdio.h>
2
3 void swap_sbagliato(int a, int b) {
4     int temp = a;
5     a = b;
6     b = temp;
7 }
8
9 int main(void) {
10     int x = 5;
11     int y = 10;
12     swap_sbagliato(x, y); // Passa i valori di x e y (fotocopie)
13     printf("x = %d, y = %d\n", x, y); // Stampa x = 5, y = 10!
14     return 0;
15 }
```

Se eseguiamo questo codice, il valore di `x` e `y` nel `main` rimarrà rispettivamente 5 e 10. Questo perché la funzione lavora su copie isolate collocate nel proprio Stack Frame, che vengono distrutte non appena la funzione termina.

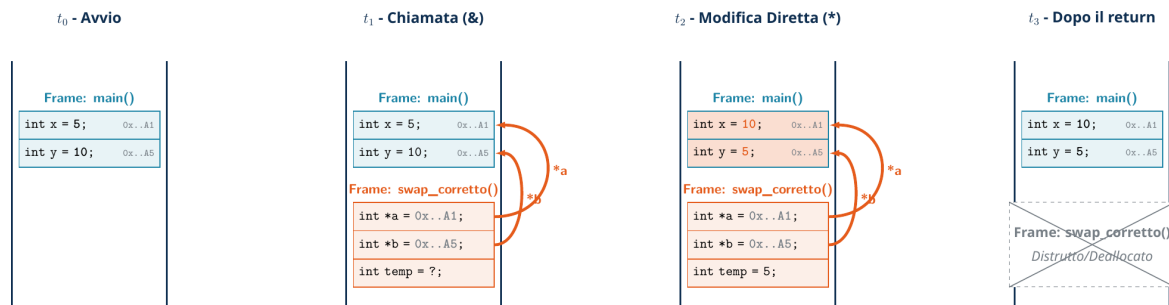


Figura 8.4: Diagramma: Swap Pass-by-Reference

8.2.2 Il passaggio per riferimento simulato

Per consentire a una funzione di modificare le variabili del chiamante, dobbiamo passarle gli **indirizzi** delle variabili (riferimento simulato tramite puntatori):

```

1  #include <stdio.h>
2
3  void swap_corretto(int *a, int *b) {
4      int temp = *a; // Copia il valore puntato da a
5      *a = *b;       // Scrive il valore puntato da b all'indirizzo a
6      *b = temp;     // Scrive temp all'indirizzo b
7  }
8
9  int main(void) {
10     int x = 5, y = 10;
11     swap_corretto(&x, &y); // Passa gli indirizzi di x e y
12     printf("x = %d, y = %d\n", x, y); // Stampa x = 10, y = 5!
13     return 0;
14 }

```

Grazie ai puntatori, swap\{\}_corretto può raggiungere direttamente la memoria riservata al main e modificare fisicamente le variabili originali.

8.2.3 La corrispondenza tra Array e Puntatori

In C esiste un legame indissolubile tra gli array e i puntatori. Quando dichiariamo un array statico:

```

1  int arr[5] = {10, 20, 30, 40, 50};

```

Il nome dell'array, `arr`, si comporta a tutti gli effetti come un **puntatore costante al suo primo elemento**. In altre parole, l'espressione `arr` è semanticamente equivalente a `\{\}&arr[0]`.

Questa corrispondenza si riflette anche nella sintassi di accesso. Quando scriviamo `arr[i]`, il compilatore traduce questa notazione in un'operazione di aritmetica dei puntatori seguita da dereferenziazione:

```

1  arr[i]  <==>  *(arr + i)

```

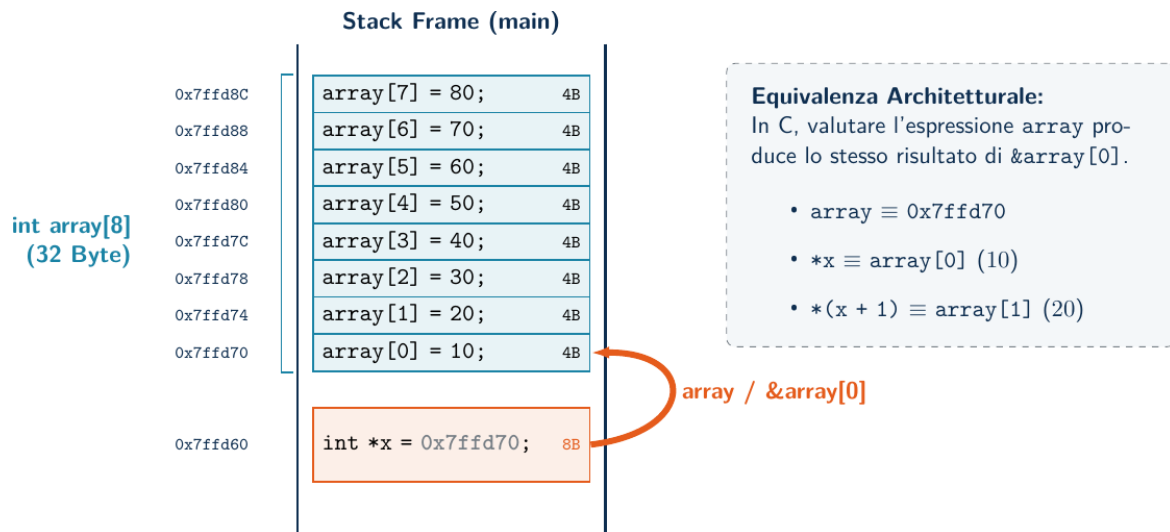


Figura 8.5: Diagramma: Equivalenza Array e Puntatori

Ciò significa che possiamo scorrere un array sia utilizzando l'indice `[i]` sia manipolando direttamente l'indirizzo tramite l'aritmetica dei puntatori.

Per chiarire questo legame intimo tra array e puntatori, osserviamo il seguente codice d'esempio:

```

1  #include <stdio.h>
2
3  int main(void) {
4      // Allocazione contigua di 8 interi (32 Byte)
5      int array[8] = {10, 20, 30, 40, 50, 60, 70, 80};
6
7      // x memorizza l'indirizzo di partenza dell'array
8      int *x = array;
9
10     // Equivalenza fondamentale:
11     // array == &array[0]
12     printf("Indirizzo array: %p\n", (void*)array);
13     printf("Indirizzo primo elemento: %p\n", (void*)&array[0]);
14     printf("Valore puntato da x: %d\n", *x); // Stampa 10
15
16     return 0;
17 }
```

A livello dello Stack Frame, le celle dell'array sono allocate in posizioni consecutive di memoria, e il puntatore `x` punta all'inizio di questa sequenza:

8.2.4 Il decadimento degli array (Array Decay)

Quando passiamo un array come parametro a una funzione, il linguaggio C non copia l'intero array in memoria (operazione che richiederebbe molto tempo e memoria per array di grandi dimensioni). Invece, applica la regola del **decadimento degli array (array decay)**: l'array decade a un semplice puntatore al suo primo elemento.

Di conseguenza, qualsiasi parametro formale dichiarato come `int arr[]` o `int arr[100]` viene interpretato dal compilatore come `int *arr`.

Questo ha un effetto collaterale importante: **la funzione perde l'informazione sulla dimensione dell'array**. Possiamo verificarlo empiricamente confrontando l'output dell'operatore `sizeof` all'interno del `main` e all'interno di una funzione:

```

1  #include <stdio.h>
2
3  void stampa_dimensione(int arr[], int dim) {
4      // All'interno della funzione, arr ? diventato un semplice puntatore
   ↪ a int (int *).
5      // sizeof(arr) restituisce la dimensione di un puntatore (8 byte su
   ↪ sistemi a 64-bit).
6      printf("In funzione: sizeof(arr) = %lu byte\n", sizeof(arr));
7  }
8
9  int main(void) {
10     int voti[5] = {18, 22, 25, 28, 30};
11
12     // Nel main, voti ? un array statico di 5 interi.
13     // sizeof(voti) restituisce la dimensione totale: 5 * sizeof(int) =
   ↪ 20 byte.
14     printf("Nel main: sizeof(voti) = %lu byte\n", sizeof(voti));
15
16     stampa_dimensione(voti, 5);
17     return 0;
18 }
```

Output del programma (su sistema a 64-bit):

```

1  Nel main: sizeof(voti) = 20 byte
2  In funzione: sizeof(arr) = 8 byte
```

Per questo motivo, in C è **sempre necessario** passare la dimensione dell'array come parametro aggiuntivo (`int dim`) a qualsiasi funzione debba elaborarlo.

8.2.5 Aritmetica dei puntatori e zucchero sintattico

Sui puntatori è possibile eseguire operazioni aritmetiche (come la somma e la sottrazione). L'aritmetica dei puntatori è **tipizzata**: aggiungere 1 a un puntatore non aumenta l'indirizzo di 1 byte, ma di una quantità pari a `sizeof(tipo)` associato al puntatore (ovvero la dimensione del tipo a cui punta).

Consideriamo il confronto nel codice seguente:

Aritmetica su Interi (`sizeof(int) == 4`):

```

1  int i[4] = {10, 20, 30, 40};
2  int *p = i; // p punta a i[0] (es. indirizzo 1000)
3
4  // p + 1 incrementa l'indirizzo di 1 * sizeof(int) = 4 byte
5  // p + 2 incrementa l'indirizzo di 2 * sizeof(int) = 8 byte
6  printf("p : %p\n", (void*)p); // 1000
7  printf("p+1 : %p\n", (void*)(p + 1)); // 1004
```

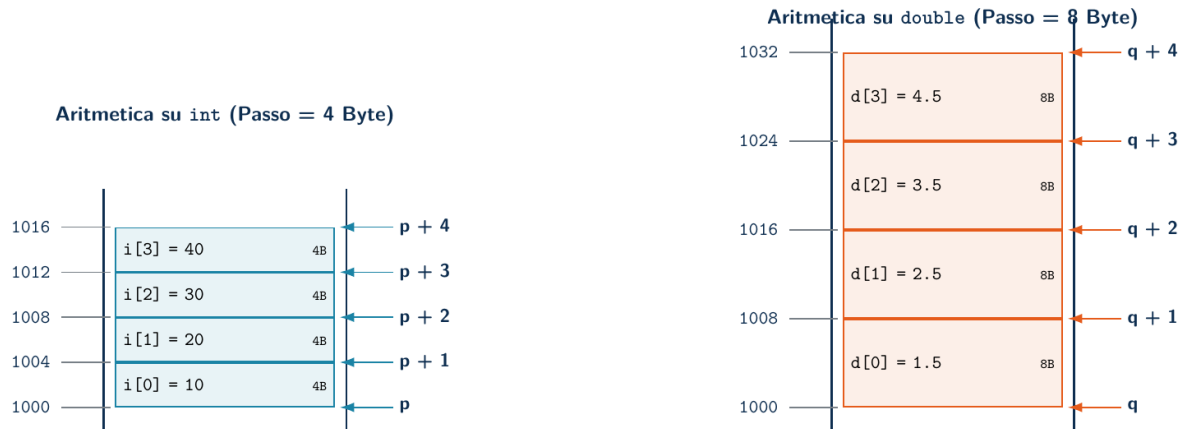



Figura 8.6: Diagramma: Aritmetica dei Puntatori

Aritmetica su Reali (sizeof(double) == 8):

```

1 double d[4] = {1.5, 2.5, 3.5, 4.5};
2 double *q = d; // q punta a d[0] (es. indirizzo 1000)
3
4 // q + 1 incrementa l'indirizzo di 1 * sizeof(double) = 8 byte
5 // q + 2 incrementa l'indirizzo di 2 * sizeof(double) = 16 byte
6 printf("q : %p\n", (void*)q); // 1000
7 printf("q+1 : %p\n", (void*)(q + 1)); // 1008

```

A livello fisico, la CPU calcola l'indirizzo sommando al valore del puntatore l'indice moltiplicato per la dimensione del tipo di dato. Il passo di incremento (passo dell'aritmetica) si adatta quindi automaticamente al tipo di puntatore:

Questo spiega perché in C l'accesso agli elementi sia estremamente veloce: si tratta di una semplice operazione aritmetica di offset seguita da dereferenziazione.

8.2.6 Sicurezza: Il modificatore const**Perché non possiamo cambiare l'indirizzo di un array**

Come abbiamo detto, il nome dell'array `arr` è un puntatore al primo elemento, ma con una restrizione cruciale: **è un puntatore costante**. Questo significa che il compilatore fissa l'indirizzo base dell'array al momento dell'allocazione e non permette di cambiarlo. Un'istruzione come la seguente genererà un errore di compilazione:

```

1 int x = 42;
2 int arr[5];
3 arr = &x; // ERRORE DI COMPILAZIONE! arr equivale a un int * const, non
           // pu? essere riassegnato

```

Proteggere i dati in lettura con const

Poiché passare un array (ovvero un puntatore al suo primo elemento) a una funzione dà a quest'ultima il potere di modificarne il contenuto originale, dobbiamo poter proteggere i dati quando vogliamo che la funzione si limiti a leggerli.

Per farlo, dichiariamo il parametro formale come puntatore a costante utilizzando la parola chiave `const`:

```
1 void stampa_array(const int *arr, int dim) {  
2     // arr[0] = 99; // ERRORE DI COMPILAZIONE! Il compilatore impedisce  
   ↪ la scrittura  
3 }
```

- `const int *p`: Il puntatore può cambiare indirizzo, ma il dato puntato è costante (sola lettura).
- `int * const p`: Il dato puntato può essere modificato, ma il puntatore non può cambiare indirizzo (comportamento nativo degli array).
- `const int * const p`: Sia il puntatore che il dato puntato sono costanti.

8.2.7 Funzioni inline

Le chiamate a funzione richiedono un costo architetturale (overhead) per la creazione e distruzione dei record di attivazione nello stack. Per funzioni piccolissime invocate frequentemente (es. all'interno di cicli intensivi), questo overhead può impattare sulle prestazioni.

La keyword `inline` suggerisce al compilatore di sostituire la chiamata con il codice effettivo direttamente nel punto in cui è invocata:

```
1 static inline int quadrato(int x) {  
2     return x * x;  
3 }
```

Questo elimina totalmente l'overhead della chiamata a funzione senza perdere i vantaggi di modularità e leggibilità del codice.

Per comprendere concretamente l'effetto dell'inlining, osserviamo a confronto il codice C e l'Assembly generato da una funzione `static inline` (in cui il compilatore ha scelto di non espandere a causa di mancate ottimizzazioni) rispetto ad una funzione classica, notando la chiamata effettiva tramite `call`.

Caso 1: Utilizzo di `static inline` (senza ottimizzazioni attive)

Se scriviamo una funzione dichiarata come `static inline`, in assenza di flag di ottimizzazione del compilatore, quest'ultimo potrebbe comunque generare una chiamata a funzione standard.

Attenzione

Perché `static inline` e non solo `inline`? In C99 una funzione dichiarata solo `inline` è una "definizione inline" che **non** produce il codice della funzione nel file oggetto: se il compilatore (ad esempio senza `-O2`) decide di non espandere la chiamata, il linker non trova la funzione e fallisce con `undefined reference to 'add'`. Aggiungendo `static` la funzione diventa locale al file e il compilatore ne genera sempre una copia

quando serve.

Codice C:

```

1  #include <stdio.h>
2
3  static inline int add(int x, int y) {
4      return x + y;
5  }
6
7  int main(void) {
8      int x = 3 + add(5, 5);
9      return x;
10 }
```

Assembly x86-64 generato per il main:

```

1  main:
2      push    rbp
3      mov     rbp, rsp
4      sub     rsp, 16
5      mov     dword ptr [rbp - 4], 0
6      mov     esi, 5
7      mov     edi, esi
8      call    add                ; Esegue comunque la chiamata alla
   ↪ funzione (locale) add
9      add     eax, 3              ; Somma 3 al risultato restituito in eax
10     mov     dword ptr [rbp - 8], eax
11     mov     eax, dword ptr [rbp - 8]
12     add     rsp, 16
13     pop     rbp
14     ret
```

Caso 2: Funzione Classica (Senza inline)

Nel caso di una funzione standard, il compilatore genera separatamente il codice della funzione add e lo invoca esplicitamente nel main.

Codice C:

```

1  #include <stdio.h>
2
3  int add(int x, int y) {
4      return x + y;
5  }
6
7  int main(void) {
8      int x = 3 + add(5, 5);
9      return x;
10 }
```

Assembly x86-64 generato:

```

1  add:
2      push    rbp
3      mov     rbp, rsp
4      mov     dword ptr [rbp - 4], edi ; Primo parametro
5      mov     dword ptr [rbp - 8], esi ; Secondo parametro
6      mov     eax, dword ptr [rbp - 4]
7      add     eax, dword ptr [rbp - 8] ; Esegue la somma
8      pop     rbp
9      ret
10
11 main:
12     push    rbp
13     mov     rbp, rsp
14     sub     rsp, 16
15     mov     dword ptr [rbp - 4], 0
16     mov     esi, 5
17     mov     edi, esi
18     call    add ; Chiama direttamente la
    ↪ funzione add
19     add     eax, 3 ; Somma 3
20     mov     dword ptr [rbp - 8], eax
21     mov     eax, dword ptr [rbp - 8]
22     add     rsp, 16
23     pop     rbp
24     ret

```

Quando invece attiviamo le ottimizzazioni del compilatore (es. `-O2`), la keyword `inline` fa sì che il codice di `add` venga iniettato direttamente all'interno di `main`, traducendo l'intera espressione in un unico calcolo immediato (con `-O2` il `main` diventa semplicemente `mov eax, 13` seguito da `ret`) ed eliminando sia l'istruzione `call` sia lo stack frame associato alla funzione chiamata.

8.3 Quiz di Auto-valutazione

Quiz 1

Domanda

1. Qual è la dimensione in byte di una variabile puntatore su un calcolatore a 64 bit?

- A) Dipende dal tipo a cui punta (es. 4 byte per int, 1 byte per char)
- B) Sempre 8 byte
- C) Sempre 4 byte

Soluzione
Risposta corretta: B
Spiegazione: Un puntatore contiene un indirizzo di memoria. Nelle architetture a 64 bit, l'indirizzo di memoria ha sempre una lunghezza fissa di 64 bit (ovvero 8 byte), indipendentemente dal tipo di dato a cui punta.

Quiz 2

Domanda

2. Quale operatore restituisce l'indirizzo di memoria di una variabile esistente?

- A) *
- B) \{\}&
- C) -->

Soluzione
Risposta corretta: B
Spiegazione: L'operatore unario di indirizzo \{\}& (address-of) estrae le coordinate di memoria fisiche in cui è allocata la variabile.

Quiz 3

Domanda

3. Se ptr è un puntatore a intero inizializzato a NULL, cosa succede se proviamo a eseguire l'istruzione *ptr = 10;?

- A) La variabile intera viene correttamente modificata
- B) Il programma viene compilato correttamente ma va in crash (Segmentation Fault) all'esecuzione
- C) Errore di compilazione immediato

Soluzione
Risposta corretta: B
Spiegazione: Compilare la dereferenziazione di un puntatore è sintatticamente valido. All'atto pratico, tentare di accedere all'indirizzo 0x0 (NULL) genera una violazione di accesso di memoria (Segmentation Fault) gestita dal sistema operativo che termina forzatamente il processo.

Quiz 4

Domanda

4. Perché i puntatori richiedono una tipizzazione (es. int* o char*)?

- A) Per definire lo spazio occupato dal puntatore stesso (8 byte vs 4 byte)
- B) Per consentire al compilatore di sapere quanti byte leggere/scrivere e come interpretarli all'indirizzo puntato
- C) È solo un vincolo estetico del linguaggio C

Soluzione
Risposta corretta: B
Spiegazione: Sebbene l'indirizzo occupi sempre 8 byte, il tipo associato specifica la "lente" attraverso cui guardare la cella: definisce quanti byte consecutivi considerare a partire da quell'indirizzo e con quale schema interpretarli.

Quiz 5

Domanda

5. Cosa significa il termine "Array Decay" in C?

- A) La progressiva perdita di precisione dei dati all'interno dell'array
- B) Il fatto che il nome di un array decade implicitamente a un puntatore al suo primo elemento quando passato come parametro
- C) La distruzione automatica dell'array alla fine del ciclo di vita

Soluzione
Risposta corretta: B
Spiegazione: Quando un array viene passato come argomento a una funzione, decade in un semplice puntatore (`\{\}\&arr[0]`), perdendo l'informazione sulla sua dimensione totale originale (`sizeof`).

Quiz 6

Domanda

6. Se un puntatore a intero p memorizza l'indirizzo decimale 2000, quale indirizzo conterrà l'espressione $p + 2$?

- A) 2002
- B) 2008
- C) 2016

Soluzione
Risposta corretta: B
Spiegazione: L'aritmetica dei puntatori scala l'addizione per la dimensione del tipo a cui punta. Poiché in genere un intero (`int`) occupa 4 byte, $p + 2$ equivale a $2000 + (2 * 4) = 2008$.

Quiz 7

Domanda

7. Qual è l'equivalenza fondamentale che lega gli array e la dereferenziazione in C?

- A) `arr[i]` equivale a `*(arr + i)`
- B) `arr[i]` equivale a `{&arr + i}`
- C) `arr[i]` equivale a `*arr + i`

Soluzione
Risposta corretta: A
Spiegazione: La notazione con parentesi quadre è solo zucchero sintattico per l'aritmetica dei puntatori. `arr[i]` esegue una somma sull'indirizzo base `arr` con offset `i` e dereferenzia il risultato.

Quiz 8

Domanda

8. Cosa dichiara l'istruzione `const int *ptr = &x;`?

- A) Un puntatore costante: non possiamo spostare `ptr` su altri indirizzi
- B) Un puntatore a un intero costante: `ptr` può puntare altrove, ma non possiamo modificare il valore di `x` tramite `ptr`
- C) Tutto bloccato: sia `ptr` che `x` sono immutabili

Soluzione
Risposta corretta: B
Spiegazione: La parola chiave `const` prima dell'asterisco protegge il valore puntato. Il puntatore stesso rimane modificabile (`ptr = &y;` è valido), ma il contenuto all'indirizzo puntato diventa in sola lettura.

Quiz 9

Domanda

9. A cosa serve la parola chiave `inline` davanti alla definizione di una funzione?

- A) A dire al compilatore di inserire il corpo della funzione direttamente nel punto di chiamata, riducendo l'overhead dello stack frame
- B) A consentire alla funzione di essere dichiarata su un'unica riga
- C) A forzare la funzione a essere compilata solo per sistemi multi-thread

Soluzione
Risposta corretta: A
Spiegazione: La keyword `inline` suggerisce al compilatore di sostituire la chiamata con il codice vero e proprio, eliminando l'overhead computazionale dovuto a push e pop del record di attivazione dello Stack.

Quiz 10

Domanda

10. Qual è il comportamento atteso dichiarando un parametro formale come `int * const arr`?

- A) L'array è di sola lettura
- B) L'indirizzo puntato da `arr` non può essere modificato (`arr` è costante), ma gli elementi all'interno possono essere cambiati
- C) Il compilatore genera un errore di sintassi

Soluzione: Collocare `const` dopo l'asterisco rende il puntatore stesso costante (sola scrittura dell'indirizzo). Gli elementi a cui punta restano accessibili in scrittura.

Risposta corretta: B

Soluzione

8.4 Esercizi Pratici

Esercizio 1: Incremento tramite Puntatore

Domanda

Scrivete una funzione `void incrementa(int *x)` che incrementi di una unità il valore contenuto all'indirizzo puntato da `x`. Nel `main`, dichiarate un intero, chiamate la funzione passandone l'indirizzo e verificate il cambiamento.

```

1  #include <stdio.h>
2
3  void incrementa(int *x) {
4      if (x != NULL) {
5          (*x)++; // Nota: le parentesi sono necessarie a causa delle
                  // precedenze
6      }
7  }
8
9  int main(void) {
10     int valore = 10;
11     printf("Valore originale: %d\n", valore);
12     incrementa(&valore);
13     printf("Valore incrementato: %d\n", valore); // Stampare? 11
14     return 0;
15 }
```

Soluzione

Esercizio 2: Prodotto di due numeri**Domanda**

Scrivete una funzione `double calcola_prodotto(const double *a, const double *b)` che calcoli e restituisca il prodotto dei due valori reali puntati da `a` e `b`. I valori originali non devono essere modificabili all'interno della funzione.

```

1  #include <stdio.h>
2
3  double calcola_prodotto(const double *a, const double *b) {
4      if (a == NULL || b == NULL)
5          return (a) * (b);
6      }
7      return 0.0;
8  }
9
10 int main(void) {
11     double n1 = 3.5;
12     double n2 = 2.0;
13     double ris = calcola_prodotto(&n1, &n2);
14     printf("Prodotto: %.2f\n", ris); // Stampa: 7.00
15     return 0;
16 }
```

Soluzione**Esercizio 3: Swap di due Double****Domanda**

Scrivete una funzione `void swap_double(double *x, double *y)` che scambi i valori reali contenuti in due variabili reali del chiamante.

```

1  #include <stdio.h>
2
3  void swap_double(double *x, double *y) {
4      if (x == NULL || y == NULL) return;
5      double temp = *x;
6      *x = *y;
7      *y = temp;
8  }
9
10 int main(void) {
11     double a = 1.23, b = 9.87;
12     printf("Prima: a=%.2f, b=%.2f\n", a, b);
13     swap_double(&a, &b);
14     printf("Dopo: a=%.2f, b=%.2f\n", a, b);
15     return 0;
16 }

```

Soluzione

Esercizio 4: Azzera Array tramite Puntatore

Domanda

Scrivete una funzione `void azzera_array(int *arr, int dim)` che azzeri tutti gli elementi di un array di interi. Utilizzate l'aritmetica dei puntatori per scorrere l'array invece degli indici quadrate (zucchero sintattico).

```

1  #include <stdio.h>
2
3  void azzera_array(int *arr, int dim) {
4      if (arr == NULL) return;
5      for (int i = 0; i < dim; i++) {
6          *(arr + i) = 0; // Uso dell'aritmetica dei puntatori
7      }
8  }
9
10 int main(void) {
11     int numeri[5] = {1, 2, 3, 4, 5};
12     azzera_array(numeri, 5);
13     for (int i = 0; i < 5; i++) {
14         printf("%d ", numeri[i]); // Stampare: 0 0 0 0 0
15     }
16     printf("\n");
17     return 0;
18 }

```

Soluzione

Esercizio 5: Riferimento a minimo (Doppio Puntatore)**Domanda**

Scrivete una funzione `void trova_min_ptr(const int *arr, int dim, const int **min_ptr)` che trovi l'indirizzo dell'elemento minimo all'interno di un array e scriva tale indirizzo nella variabile puntata da `min_ptr`.

```

1  #include <stdio.h>
2
3  void trova_min_ptr(const int *arr, int dim, const int **min_ptr) {
4      if (arr == NULL || min_ptr == NULL || dim <= 0) return;
5
6      const int *min_corrente = arr; // Inizializza al primo elemento
7      for (int i = 1; i < dim; i++) {
8          if (*(arr + i) < *min_corrente)
9              min_corrente = arr + i;
10     }
11     *min_ptr = min_corrente; // Scrive l'indirizzo trovato
12 }
13
14 int main(void) {
15     int dati[5] = {34, 12, 56, 8, 45};
16     const int *min_addr = NULL;
17
18     trova_min_ptr(dati, 5, &min_addr);
19
20     if (min_addr != NULL)
21         printf("Il minimo e': %d, all'indirizzo %p\n", *min_addr, (
22         void *)min_addr);
23     }
24     return 0;
25 }

```

Soluzione**Esercizio 6: Inversione di un array in-place con due****Domanda**

puntatori

Scrivete una funzione `void inverti_array(int *arr, int dim)` che inverta gli elementi di un array sul posto usando due puntatori di supporto: uno impostato al primo elemento, l'altro all'ultimo elemento. Si effettui lo swap degli elementi e si muovano i puntatori l'uno verso l'altro finché non si incontrano.

```

1  #include <stdio.h>
2
3  void inverti_array(int *arr, int dim)
4  {
5      if (arr == NULL || dim <= 1) return;
6
7      int *sinistra = arr;
8      int *destra = arr + dim - 1;
9
10     while (sinistra < destra)
11     {
12         // Swap degli elementi puntati
13         int temp = *sinistra;
14         *sinistra = *destra;
15         *destra = temp;
16
17         // Spostamento dei puntatori
18         sinistra++;
19         destra--;
20     }
21
22     int v[6] = {10, 20, 30, 40, 50, 60};
23     inverti_array(v, 6);
24     for (int i = 0; i < 6; i++) {
25         printf("%d ", v[i]); // Stampare? 60 50 40 30 20 10
26     }
27     printf("\n");
28     return 0;
29 }

```

Soluzione**Esercizio 7: Confronto di Puntatori****Domanda**

Scrivete una funzione `int confronta({}_puntatori(const int *p1, const int *p2)` che verifichi i valori puntati. Restituisca 1 se il valore puntato da `p1` è maggiore di quello di `p2`, -1 se è minore, e 0 se sono uguali. Se uno dei due puntatori è nullo, la funzione deve restituire -999.

```

1  #include <stdio.h>
2
3  int confronta_puntatori(const int *p1, const int *p2) {
4      if (p1 == NULL || p2 == NULL)
5          return -999;
6      }
7      if (*p2 < *p1)
8          return 1;
9      if (*p2 > *p1)
10         return -1;
11     return 0;
12 }
13
14 int main(void) {
15     int a = 15, b = 20;
16     printf("Risultato: %d\n", confronta_puntatori(&a, &b)); //
17     // Stampare? -1
18     printf("Risultato NULL: %d\n", confronta_puntatori(&a, NULL));
19     // Stampare? -999
20     return 0;
21 }

```

Soluzione**Esercizio 8: Lunghezza stringa con Aritmetica dei****Domanda****Puntatori**

Scrivete una funzione `int calcola_lunghezza_stringa(const char *s)` che calcoli la lunghezza di una stringa in C (terminata dal carattere `\0`) incrementando un puntatore da inizio stringa e calcolando la differenza di indirizzi alla fine. Non utilizzate indici o variabili di conteggio.

```

1  #include <stdio.h>
2
3  int calcola_lunghezza_stringa(const char *s) {
4      if (s == NULL) return 0;
5      const char *p = s;
6      while (*p != '\0') {
7          p++; // Avanza il puntatore
8      }
9      return p - s; // Differenza tra puntatori: restituisce il
10     } // numero di char
11
12 int main(void) {
13     char testo[] = "Puntatori";
14     printf("Lunghezza di '%s': %d\n", testo,
15     ↪ calcola_lunghezza_stringa(testo)); // Stampare? 9
16     return 0;
17 }

```

Soluzione**Esercizio 9: Copia Sicura di un Array Constant****Domanda**

Scrivete una funzione `void copia\{\}_array\{\}_const(const int *sorgente, int *destinazione, int dim)` che copi tutti gli elementi da un array sorgente (in sola lettura) a un array di destinazione usando puntatori.

```

1  #include <stdio.h>
2
3  void copia_array_const(const int *sorgente, int *destinazione, int
4      ↵ dim)
5  {
6      if (sorgente == NULL || destinazione == NULL) return;
7      for (int i = 0; i < dim; i++)
8          *(destinazione + i) = *(sorgente + i);
9  }
10 int main(void) {
11     int src[4] = {5, 10, 15, 20};
12     int dst[4];
13     copia_array_const(src, dst, 4);
14     for (int i = 0; i < 4; i++)
15         printf("%d ", dst[i]); // Stampare? 5 10 15 20
16     }
17     printf("\n");
18     return 0;
19 }

```

Soluzione**Esercizio 10: Swap di Puntatori****Domanda**

Scrivete una funzione `void swap\{\}_puntatori(int **p1, int **p2)` che scambi gli indirizzi di memoria memorizzati all'interno di due variabili puntatore. Nel main, verificate che dopo lo swap, i due puntatori puntino a variabili inverse.

Soluzione

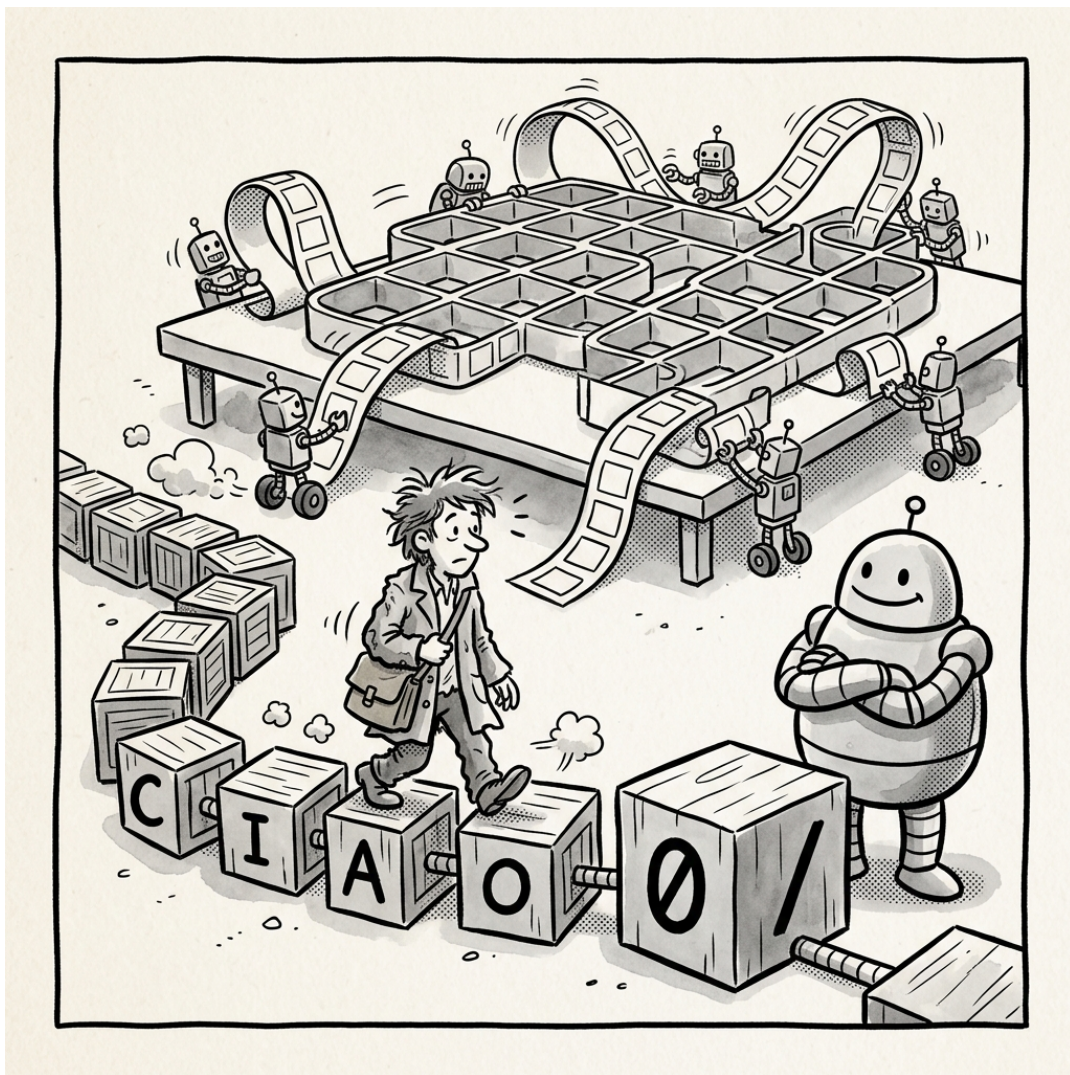
```

1  #include <stdio.h>
2
3  void swap_puntatori(int **p1, int **p2) {
4      if (p1 != NULL && p2 != NULL) {
5          int *temp = *p1;
6          *p1 = *p2;
7          *p2 = temp;
8      }
9  }
10
11 int main(void) {
12     int x = 100, y = 200;
13     int *p1 = &x;
14     int *p2 = &y;
15
16     printf("p1 punta a valore: %d, p2 punta a valore: %d\n", *
17         ↪ p1, *p2);
18     swap_puntatori(&p1, &p2); // Scambia gli indirizzi
19         ↪ contenuti in p1 e p2
20     printf("Dopo swap:\n");
21     printf("p1 punta a valore: %d, p2 punta a valore: %d\n", *
22         ↪ p1, *p2);
23     // Stampa? p1 punta a valore 200 e p2 punta a valore 100
24     return 0;
25 }

```


LEZIONE 9

Le Stringhe e lo Spazio Multidimensionale



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 9

Le Stringhe e lo Spazio Multidimensionale

In questa lezione affronteremo due concetti apparentemente distanti ma accomunati dallo stesso principio architettonico: **come la memoria RAM (struttura lineare e contigua) viene organizzata dal compilatore per rappresentare astrazioni complesse come testi (stringhe) e griglie geometriche (matrici)**. Comprendremo perché il C non possiede un tipo stringa nativo, l'importanza della sentinella `\{\}\{\}\0` per delimitare i dati utili all'interno di buffer statici, e come calcolare gli indirizzi fisici degli elementi multidimensionali in Row-Major Order.

9.1 L'Anatomia delle Stringhe e il Terminatore Null

9.1.1 L'Astrazione del Testo (Il tipo che non esiste)

Fino a questo momento abbiamo manipolato tipi numerici primitivi (`int`, `float`, `double`) e singoli caratteri alfanumerici (`char`). Tuttavia, per gestire una sequenza di caratteri (parole, frasi, percorsi file), ci accorgiamo che **in C non esiste un tipo di dato nativo string**.

Fedele alla sua natura di linguaggio orientato al basso livello, il C rappresenta una stringa come una sequenza ordinata e contigua di caratteri in memoria. Ovvero: **un semplice array di caratteri (`char`)**.

```
1 char parola[10];
```

Poiché l'array ha una dimensione fissa allocata staticamente nello Stack (nell'esempio sopra, 10 byte), sorge un problema: i testi che scriviamo hanno lunghezze variabili (es. "Etna" ha 4 caratteri, "Catania" ne ha 7). Come fa il programma a sapere dove finisce il testo utile all'interno dell'array senza "sbrodolare" nelle celle adiacenti dello Stack?

9.1.2 Inizializzazione e l'Importanza del Terminatore `\0`

In C, la fine logica di una stringa è marcata da un carattere speciale chiamato **sentinella** o **terminatore null**, rappresentato dalla sequenza di escape `\{\}\{\}\0` (corrispondente al valore binario 0 della tabella ASCII).

Possiamo inizializzare una stringa in due modi:

```

1 // Metodo 1: Inizializzazione Implicita (Virgolette double quotes)
2 // Il compilatore alloca automaticamente 5 byte (4 caratteri + il '\0'
  ↪ inserito in coda)
3 char s1[] = "Ciao";
4
5 // Metodo 2: Inizializzazione Esplicita (Richiede l'inserimento manuale
  ↪ del terminatore)
6 char s2[] = {'C', 'i', 'a', 'o', '\0'};

```

Cosa succede se dimentichiamo il `\0`?

Consideriamo questo frammento di codice errato:

```

1 #include <stdio.h>
2
3 int main(void) {
4     char s_bug[] = {'B', 'u', 'g'}; // ERRORE: Manca la sentinella!
5     printf("%s\n", s_bug);
6     return 0;
7 }

```

Quando invochiamo `printf("\{%s", s\{}_bug)`, la CPU si posiziona sull'indirizzo iniziale dell'array `s\{}_bug` (la lettera 'B') e inizia a stampare consecutivamente i caratteri in RAM. Poiché non trova il valore nullo `\{}\{}\0` che gli impone l'arresto, la CPU continuerà a leggere e interpretare come caratteri le celle di memoria adiacenti sullo Stack (dangling memory). Questo comporta la stampa a video di simboli casuali e spazzatura finché, per pura coincidenza, non incontra un byte a zero nella RAM, oppure causa un crash immediato per violazione di accesso (Segmentation Fault).

9.1.3 Lunghezza Massima vs Lunghezza Logica

Grazie alla sentinella `\{}\{}\0`, possiamo riservare un buffer statico molto capiente (es. `char nome[100]`) e utilizzarlo per memorizzare stringhe di lunghezza variabile.

- **Dimensione Statica (Lunghezza Massima):** è la memoria fisicamente riservata dal compilatore nello Stack (il numero di elementi dell'array).
- **Lunghezza Logica (Dimensione Effettiva):** è il numero di caratteri utili presenti prima del terminatore `\{}\{}\0` (misurato dalla funzione `strlen`).

Il seguente diagramma mostra chiaramente la differenza strutturale tra due stringhe in memoria a parità di dimensione allocata (10 Byte):

I caratteri successivi alla sentinella `\{}\{}\0` sono **caratteri spuri (spazzatura)** e vengono completamente ignorati dalle funzioni standard di stampa e manipolazione del linguaggio C.

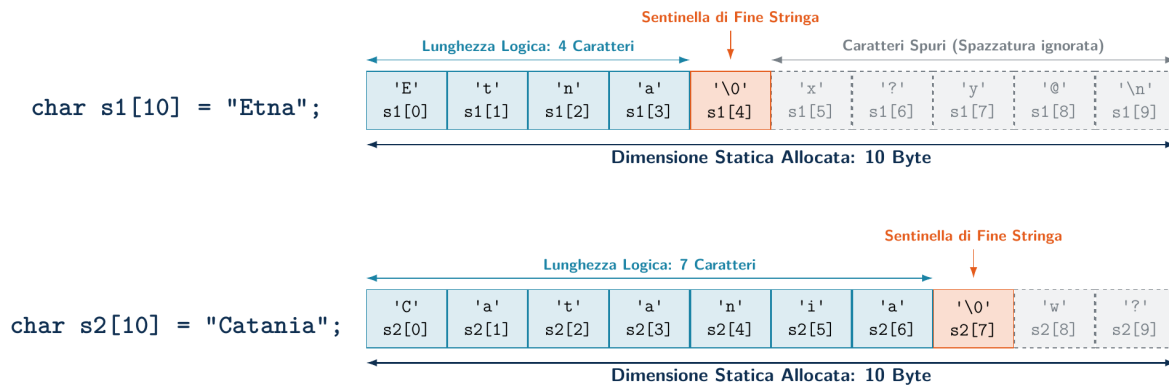


Figura 9.1: Diagramma: Anatomia delle Stringhe

9.1.4 Scansione di Stringhe ed Aritmetica dei Puntatori

Vediamo due tecniche per scansionare una stringa carattere per carattere fino al raggiungimento del terminatore.

```

1  #include <stdio.h>
2
3  int main(void) {
4      char str[] = "Etna";
5
6      // Versione A: Indicizzazione classica dell'array
7      printf("Versione A: ");
8      for (int i = 0; str[i] != '\0'; i++) {
9          putchar(str[i]);
10     }
11     putchar('\n');
12
13     // Versione B: Scansione con puntatori (Aritmetica dei puntatori)
14     printf("Versione B: ");
15     char *p = str; // p punta all'indirizzo iniziale str[0]
16     while (*p != '\0') {
17         putchar(*p);
18         p++; // Sposta l'indirizzo avanti di 1 byte (dimensione di un
19         ↪ char)
20     }
21     putchar('\n');
22
23     return 0;
24 }
```

Tagliare le stringhe con l'aritmetica

Poiché il nome dell'array `str` decade a puntatore al primo elemento, possiamo stampare sotto-stringhe senza copiare dati in nuovi array semplicemente variando l'indirizzo di partenza:

```

1  printf("%s\n", str + 2); // Salta i primi due caratteri e stampa "na"
```

9.2 Manipolazione, Librerie Standard e Sicurezza

9.2.1 Analisi dei Caratteri con <ctype.h>

Per evitare l'uso di "numeri magici" legati alla tabella ASCII (es. `if (c >= 65 && c <= 90)` per individuare lettere maiuscole), il C mette a disposizione la libreria standard <ctype.h>. Questa contiene funzioni ottimizzate ed indipendenti dal set di caratteri utilizzato:

```

1  #include <stdio.h>
2  #include <ctype.h>
3
4  int main(void) {
5      char c = 'a';
6
7      if (isalpha(c)) {
8          printf("%c e' una lettera\n", c);
9      }
10     if (islower(c)) {
11         printf("Ed e' minuscola! In maiuscolo: '%c'\n", toupper(c));
12     }
13     return 0;
14 }
```

Principali funzioni di <ctype.h>:

- `isalpha(c)`: restituisce vero se `c` è una lettera.
- `isdigit(c)`: restituisce vero se `c` è una cifra decimale (0..9).
- `isalnum(c)`: restituisce vero se `c` è alfanumerico.
- `isspace(c)`: restituisce vero se `c` è uno spazio, tabulazione o ritorno a capo.
- `toupper(c)` / `tolower(c)`: converte il carattere.

9.2.2 Conversione Sicura di Stringhe (strtod vs atoi)

Quando acquisiamo dati testuali, abbiamo spesso la necessità di convertirli in valori numerici per effettuare operazioni matematiche.

Approccio Unsafe: atoi (e atof)

La funzione `atoi` (ASCII to Integer) della libreria <stdlib.h> è molto diffusa ma presenta un grave difetto di sicurezza:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5      int val = atoi("123"); // Ritorna 123 (Corretto)
6      int err = atoi("ciao"); // Ritorna 0! (Fallimento)
7  }
```



```

7   char buffer[50];
8
9   // Lunghezza logica
10  printf("Lunghezza di str1: %zu\n", strlen(str1)); // Stampa 4
11
12  // Copia
13  strcpy(buffer, str1);
14  printf("Buffer copiato: %s\n", buffer); // Stampa "Etna"
15
16  // Concatenazione
17  strcat(buffer, str2);
18  printf("Buffer concatenato: %s\n", buffer); // Stampa "Etna Coder"
19
20  // Confronto lessicografico
21  if (strcmp(str1, "Etna") == 0) {
22      printf("Le due stringhe sono identiche!\n");
23  }
24
25  int confronto = strcmp("albero", "barca");
26  printf("Confronto 'albero' vs 'barca': %d\n", confronto); // Ritorna
    ↪ valore < 0
27
28  return 0;
29 }

```

9.3 L'Input/Output di Testo Sicuro e le Matrici

9.3.1 La piaga del Buffer Overflow e la soluzione fgets

L'acquisizione classica di stringhe tramite `scanf("\{%s", buffer)` costituisce una delle principali vulnerabilità software storiche. Se `buffer` ha una dimensione limitata (es. 10 byte) e l'utente inserisce una parola di 30 caratteri, `scanf` continuerà a scrivere in RAM sovrascrivendo i record adiacenti e gli indirizzi di ritorno dello Stack Frame (Buffer Overflow).

Esempio Unsafe (Che manderemo in crash):

```

1  #include <stdio.h>
2
3  int main(void) {
4      char password[8] = "segreto"; // 7 caratteri + '\0'
5      char input[8];                // Buffer di input limitato a 8 byte
6
7      printf("Inserisci la password di sblocco: ");
8      // PERICOLO: scanf non sa quanto sia grande 'input'!
9      scanf("%s", input);
10
11     // Se l'utente digita una password molto lunga (es. "1234567890abcdef
    ↪ "),
12     // i caratteri in eccesso sborderanno in memoria sovrascrivendo
13     // la variabile 'password' o i registri dello Stack Frame.
14     printf("Hai inserito: %s\n", input);

```

```

15     printf("Password di controllo attuale: %s\n", password);
16
17     return 0;
18 }

```

Se provate ad eseguire questo codice compilato ed inserite una stringa molto lunga, vedrete che la variabile `password` verrà alterata dal vostro stesso input, o il sistema andrà in crash restituendo un errore del tipo `Illegal instruction` o `Segmentation Fault`.

Per risolvere questo problema si deve utilizzare **`fgets`**, che limita esplicitamente i byte letti:

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void) {
5      char nome[20];
6      printf("Inserisci il tuo nome: ");
7
8      // Legge al massimo 19 caratteri lasciando il 20-esimo per il '\0'
9      if (fgets(nome, sizeof(nome), stdin) != NULL) {
10         // fgets acquisisce anche l'invio '\n', rimuoviamolo se presente
11         nome[strcspn(nome, "\n")] = '\0';
12         printf("Ciao %s!\n", nome);
13     }
14     return 0;
15 }

```

Svuotare il buffer residuo

Se leggiamo un valore numerico con `scanf` e subito dopo proviamo ad acquisire una stringa con `fgets`, quest'ultima leggerà l'invio (`\{\}\{\}\n`) rimasto intrappolato nel buffer di input di sistema. Per pulire il buffer residuo prima di una `fgets`:

```

1  int c;
2  while ((c = getchar()) != '\n' && c != EOF); // Consuma i caratteri spuri

```

9.3.2 Le Matrici (Linearizzazione in Row-Major Order)

Dal punto di vista logico, immaginiamo una matrice come una griglia bidimensionale `int M[Righe][Colonne]`. In realtà, **la memoria RAM è rigorosamente lineare (unidimensionale)**.

Il C risolve questa astrazione implementando il **Row-Major Order**: le righe della matrice vengono accodate consecutivamente in RAM.

Per calcolare l'indirizzo fisico in cui risiede l'elemento generico `M[i][j]`, la CPU applica la seguente formula di offset:

$$\text{Indirizzo Fisico} = \text{Indirizzo Base} + (i \times \text{Numero Colonne} + j) \times \text{sizeof(tipo)}$$

Notiamo che la CPU **non ha bisogno di conoscere il numero di righe complessivo**, ma deve necessariamente conoscere il **numero di colonne** per poter calcolare quanti elementi scavalcare per posizionarsi alla riga desiderata.

Rappresentazione Logica (Griglia 3×4)

Riga 0	M[0][0] (1)	M[0][1] (2)	M[0][2] (3)	M[0][3] (4)
Riga 1	M[1][0] (5)	M[1][1] (6)	M[1][2] (7)	M[1][3] (8)
Riga 2	M[2][0] (9)	M[2][1] (10)	M[2][2] (11)	M[2][3] (12)

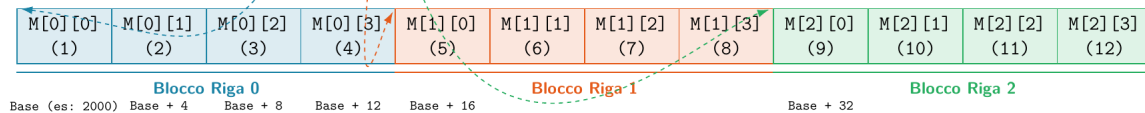
Rappresentazione Fisica in RAM (Disposizione Lineare)

Figura 9.2: Diagramma: Linearizzazione delle Matrici

Ecco perché nella firma delle funzioni che accettano matrici è obbligatorio specificare la dimensione delle colonne:

```

1  #include <stdio.h>
2
3  // La colonna (4) serve alla CPU per il calcolo corretto dell'offset in
   ⇨ RAM!
4  void stampa_matrice(int M[][4], int righe) {
5      for (int i = 0; i < righe; i++) {
6          for (int j = 0; j < 4; j++) {
7              printf("%d\t", M[i][j]);
8          }
9          printf("\n");
10     }
11 }
```

9.3.3 Matrici di Caratteri come Array di Stringhe

Una matrice bidimensionale di caratteri (`char arr[Righe][Colonne]`) può essere utilizzata in C come un **array di stringhe**. In questa astrazione, ciascuna riga rappresenta una stringa indipendente (il cui termine logico è stabilito dalla sentinella `\{\}\{\}\0`), mentre la dimensione delle colonne definisce la capacità fisica massima riservata per ogni elemento.

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void) {
5      // Definizione di un array di stringhe: 3 elementi, ciascuno con
   ⇨ spazio per massimo 10 byte
6      char arr[3][10] = {
7          "Etna",
8          "Corso",
9          "Catania"
10     };
11 }
```

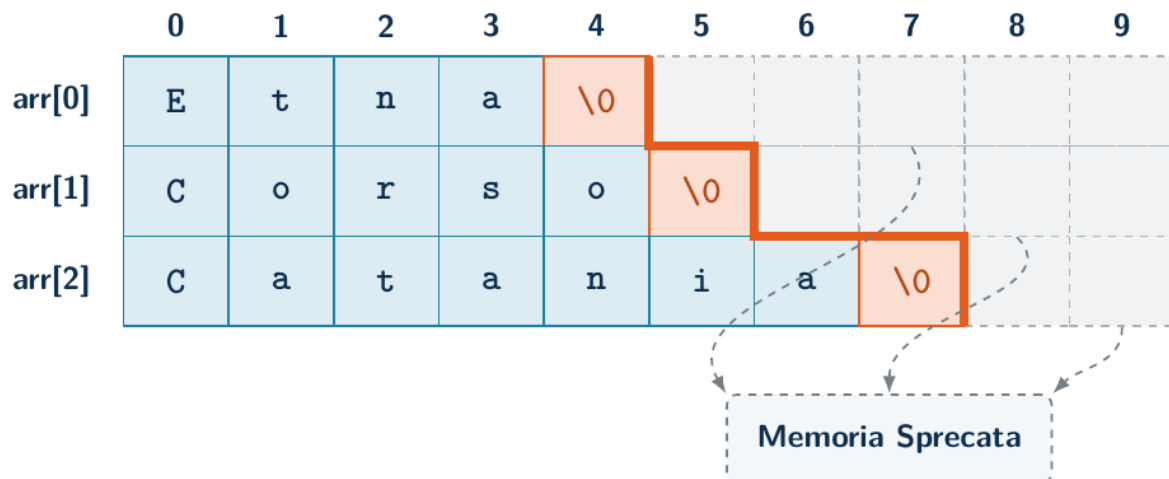


Figura 9.3: Diagramma: Rappresentazione in memoria di un array di stringhe e spreco di spazio

```

12 // arr[i] rappresenta l'indirizzo di inizio della riga i-esima (
    ↪ quindi della stringa corrispondente)
13 for (int i = 0; i < 3; i++) {
14     printf("Indice %d: %s\n", i, arr[i]);
15 }
16 return 0;
17 }

```

Rappresentazione in Memoria e lo Spreco di Spazio

Essendo la griglia statica della matrice fissa in memoria Stack, dichiarare `char arr[3][10]` impone al compilatore di allocare rigidamente $3 \times 10 = 30$ byte. Se una stringa memorizzata è più corta del limite massimo (ad esempio "Etna" che occupa solo 5 byte totali, incluso `\{\}\{\}\0`), i rimanenti byte di quella riga rimarranno allocati ma inutilizzati, causando uno spreco di spazio.

Operazioni Invalide sugli Array di Stringhe

Dato che `arr[0]` rappresenta l'indirizzo base di un array fisso (quindi un puntatore costante non riassegnabile), è **impossibile cambiare il valore di una stringa con una riassegnazione diretta**:

```

1 char arr[3][10] = {"Etna", "Corso", "Catania"};
2 // ERRORE: "assignment to expression with array type"
3 arr[0] = "GFG";

```

Per modificare i testi memorizzati all'interno dell'array di stringhe dobbiamo ricorrere alla copia profonda dei caratteri tramite `strcpy`:

```

1 // Operazione Corretta: copia "GFG" (compreso terminatore) nel buffer arr
    ↪ [0]
2 strcpy(arr[0], "GFG");

```

Nelle lezioni successive vedremo come ovviare a questo spreco di memoria e abilitare la riassegnazione diretta dei puntatori impiegando gli **array di puntatori a stringhe** (`char *arr[]`).

9.3.4 Array N-Dimensionali

Poiché le dimensioni superiori sono solo astrazioni mentali, possiamo definire array a qualunque profondità desiderata. Per la CPU si tratterà sempre di una sequenza piatta in RAM.

Consideriamo un monitoraggio termodinamico di un plesso scolastico:

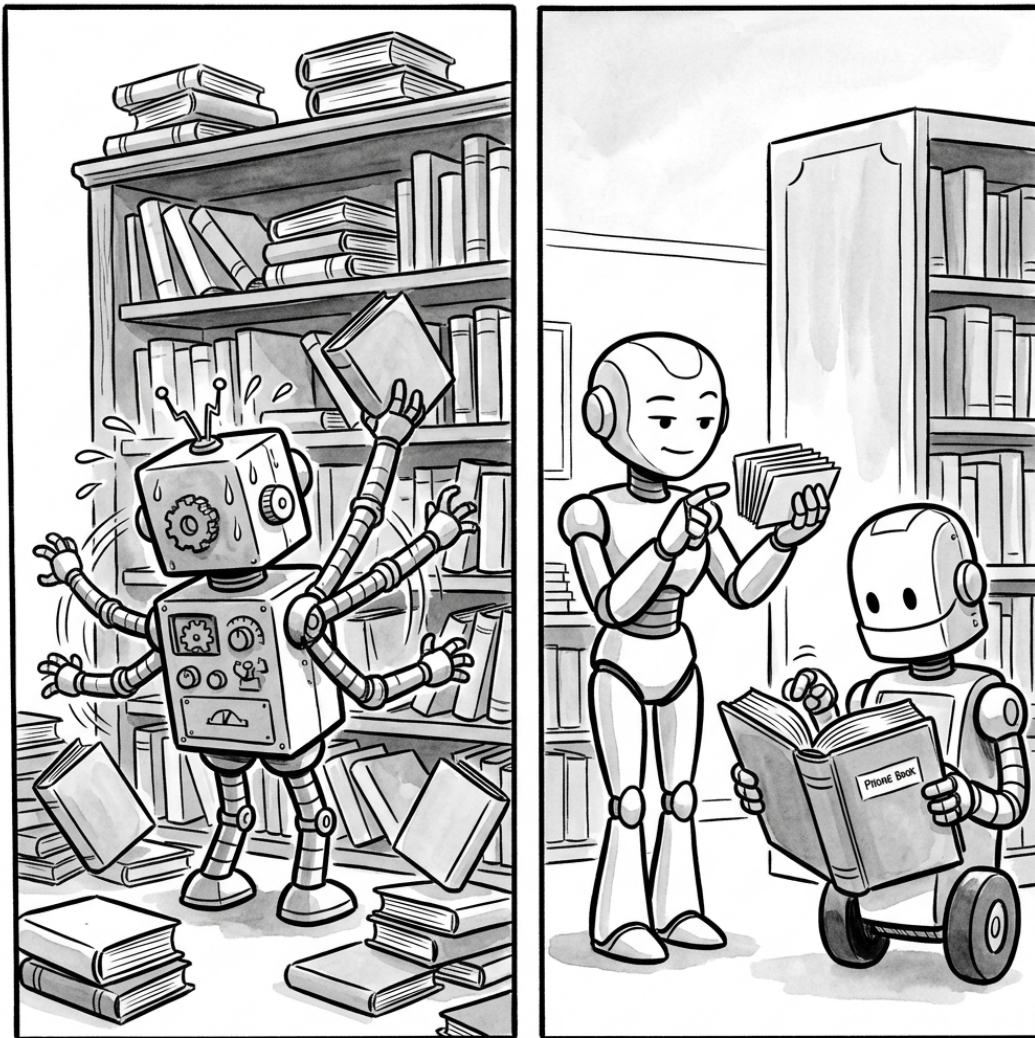
```
1 // 1D: Fila di 10 sensori lungo un tubo -> double tubo[10];
2 // 2D: Griglia di 10x10 sensori su un pavimento -> double pavimento
  ⇨ [10][10];
3 // 3D: Reticolo 10x10x5 sensori in una stanza volumetrica -> double
  ⇨ stanza[10][10][5];
4 // 4D: Misura 3D ad ogni ora di un giorno (24 ore) -> double log
  ⇨ [24][10][10][5];
5
6 // 5D: Monitoriamo 8 stanze diverse dello stesso plesso
7 double plesso[8][24][10][10][5];
```

Per accedere alla temperatura registrata nella Stanza 2, alle ore 14:00, alle coordinate spaziali $X = 4, Y = 7, Z = 1$:

```
1 double temp = plesso[2][14][4][7][1];
```

La CPU risolverà questo indirizzo espandendo ricorsivamente la formula di offset basandosi sulle dimensioni di ciascun blocco allocate a compile-time.

Algoritmi Iterativi di Ordinamento e Ricerca



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 10

Algoritmi Iterativi di Ordinamento e Ricerca

Nelle lezioni precedenti abbiamo visto come allocare, scansionare e manipolare dati in memoria RAM organizzandoli in strutture sequenziali (gli array monodimensionali e le matrici multidimensionali). Tuttavia, la vera utilità dei dati risiede nella nostra capacità di elaborarli in modo efficiente. In questa lezione affronteremo uno dei pilastri fondamentali dell'informatica: **l'ordinamento dei dati**. Impareremo tre algoritmi iterativi classici (**Selection Sort**, **Bubble Sort** e **Insertion Sort**) analizzando come lavorano sui puntatori e indici a livello fisico, e scopriremo come l'ordinamento sia la chiave per abilitare la **Ricerca Dicotomica (Binary Search)**, una delle ottimizzazioni algoritmiche più potenti ed eleganti della programmazione.

10.1 Il Ruolo dell'Ordinamento e lo Swap

10.1.1 Perché Ordiniamo i Dati?

Per comprendere l'importanza dell'ordinamento, immaginiamo uno scenario reale: cercare un nominativo specifico all'interno di un elenco telefonico di un milione di persone.

- Se l'elenco fosse **disordinato**, non avremmo altra scelta se non quella di esaminare i nomi uno ad uno, dal primo all'ultimo. Nel caso peggiore (nominativo assente o all'ultima pagina), dovremmo effettuare esattamente **1 milione di confronti**.
- Se l'elenco è **ordinato alfabeticamente**, possiamo individuare il nome in pochissimi secondi aprendo l'elenco a metà, scartando la porzione non utile e ripetendo il processo. Come vedremo, per un milione di persone basteranno al massimo **20 confronti**.

L'ordinamento è quindi la preconditione fondamentale che trasforma la ricerca di un'informazione da un processo estenuante a una frazione infinitesima di secondo.

10.1.2 La Funzione swap con Puntatori

Tutti gli algoritmi di ordinamento che analizzeremo si basano su un'operazione fondamentale: lo scambio (*swap*) di posizione tra due elementi in memoria. Per evitare che la copia sovrascriva un valore cancellandolo, dobbiamo utilizzare una variabile temporanea di appoggio. Sfruttando i puntatori (come introdotto nella Lezione 8), la funzione si presenta così:

```
1 void swap(int *a, int *b) {  
2     int temp = *a; // Salva il valore puntato da a in una cassaforte  
    ↪ temporanea  
3     *a = *b;      // Copia il valore puntato da b nell'indirizzo di a  
4     *b = temp;    // Sposta il valore temporaneo nell'indirizzo di b  
5 }
```

10.2 Il Mescolamento Casuale: Fisher-Yates Shuffle

Prima di affrontare il problema di mettere in ordine un array, consideriamo il problema opposto: **mescolarlo in modo completamente casuale ed equo**. Questa operazione, nota come *shuffling*, è fondamentale in ambiti quali simulazioni, videogiochi (es. mescolare un mazzo di carte) o generazione di chiavi di cifratura.

10.2.1 Perché lo Shuffling “Ingenuo” è Sbagliato

Un approccio istintivo potrebbe essere quello di scorrere l'array e scambiare ogni elemento con un altro scelto a caso tra tutti gli indici dell'array:

```
1 // Algoritmo errato / distorto!  
2 for (int i = 0; i < n; i++) {  
3     int j = rand() % n; // Scelta tra 0 e n-1  
4     swap(&arr[i], &arr[j]);  
5 }
```

Sebbene sembri corretto, questo metodo genera permutazioni **non uniformi**. In un array di N elementi, ci sono $N!$ permutazioni possibili. L'algoritmo sopra effettua N scelte, ciascuna con N possibilità, per un totale di N^N possibili percorsi. Poiché N^N non è quasi mai divisibile per $N!$, alcune permutazioni saranno statisticamente più frequenti di altre, introducendo una forte distorsione (bias).

10.2.2 L'Algoritmo di Fisher-Yates

L'algoritmo di **Fisher-Yates** (nella versione moderna di Durstenfeld) risolve questo problema riducendo progressivamente lo spazio di scelta.

1. Si posiziona una barriera virtuale che inizialmente esclude l'intero array.
2. Si parte dall'ultimo elemento dell'array (indice $i = n - 1$).
3. Si sceglie un indice casuale j compreso tra 0 e i (incluso).
4. Si scambiano gli elementi in posizione i e j .
5. Si decrementa i di uno e si ripete il processo fino all'indice 1.

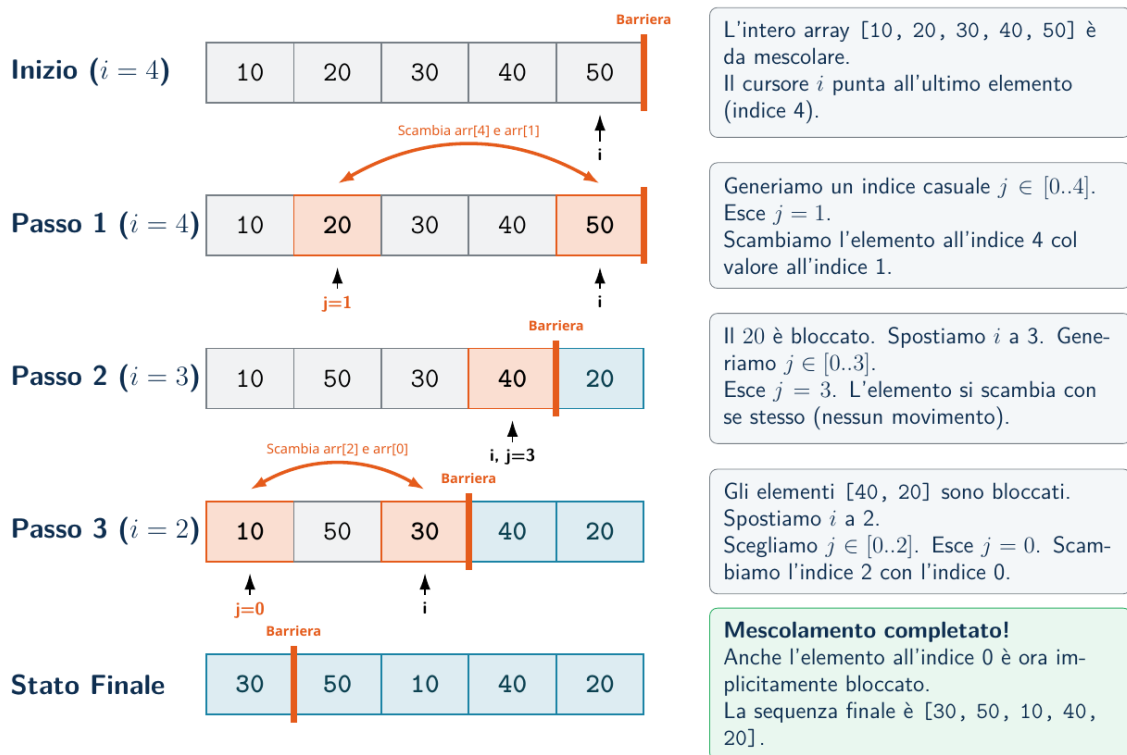


Figura 10.1: Algoritmo Fisher-Yates in azione

A ogni passo l'elemento scambiato finisce a destra della barriera rossa ed è bloccato; lo spazio di scelta si restringe esattamente a $i + 1$ elementi. Poiché il numero di percorsi possibili è esattamente $N \cdot (N - 1) \cdot \dots \cdot 2 \cdot 1 = N!$, ogni permutazione ha esattamente probabilità $1/N!$ di verificarsi. L'algoritmo opera in tempo lineare $O(N)$ e *in-place* (senza memoria aggiuntiva).

Codice C del Fisher-Yates Shuffle:

```

1 void fisher_yates(int arr[], int n) {
2     for (int i = n - 1; i > 0; i--) {
3         // Sceglie un indice casuale tra 0 e i (incluso)
4         int j = rand() % (i + 1);
5
6         // Scambia solo se gli indici differiscono
7         if (i != j) {
8             swap(&arr[i], &arr[j]);
9         }
10    }
11 }

```

10.3 I Tre Algoritmi di Ordinamento Iterativi



Figura 10.2: Selection Sort in Azione

10.3.1 Selection Sort: “Trova il Minimo”

La strategia del **Selection Sort** è estremamente intuitiva:

1. Immaginiamo una barriera logica che separa l'array in due parti: a sinistra gli elementi già ordinati, a destra quelli ancora disordinati. All'inizio l'intero array è disordinato.
2. Scansioniamo l'area disordinata alla ricerca del valore minimo.
3. Scambiamo questo valore minimo con l'elemento che si trova all'estrema sinistra dell'area disordinata.
4. Spostiamo la barriera di un passo verso destra e ripetiamo il processo per la parte rimanente dell'array.

Codice C Completo del Selection Sort:

```

1  #include <stdio.h>
2
3  void swap(int *a, int *b) {
4      int temp = *a;
5      *a = *b;
6      *b = temp;
7  }
8
9  void selection_sort(int arr[], int n) {
10     // La barriera 'i' avanza fino al penultimo elemento dell'array
11     for (int i = 0; i < n - 1; i++) {
12         int min_idx = i; // Assumiamo temporaneamente che il minimo sia
13         // all'indice 'i'
14
15         // Scansione della sotto-area disordinata (a destra di 'i')
16         for (int j = i + 1; j < n; j++) {

```



```

16         if (arr[j] < arr[min_idx]) {
17             min_idx = j; // Trovato un nuovo minimo, memorizziamo l'
           ↪ indice
18         }
19     }
20
21     // Se il minimo identificato non coincide con la testa corrente,
           ↪ scambiamo
22     if (min_idx != i) {
23         swap(&arr[i], &arr[min_idx]);
24     }
25 }
26 }
27
28 int main(void) {
29     int dati[] = {4, 2, 5, 1, 3};
30     int n = 5;
31
32     selection_sort(dati, n);
33
34     printf("Selection Sort: ");
35     for(int i = 0; i < n; i++) {
36         printf("%d ", dati[i]);
37     }
38     printf("\n");
39     return 0;
40 }

```

10.3.2 Bubble Sort: “Le Bollicine in RAM”

L'idea fondamentale del **Bubble Sort** è far risalire gli elementi più grandi verso l'estremità destra dell'array (come bollicine di anidride carbonica in un bicchiere d'acqua):

1. Si confronta ciascun elemento con il suo adiacente destro ($arr[j]$ con $arr[j+1]$).
2. Se l'elemento di sinistra è maggiore di quello di destra, vengono scambiati.
3. Al termine della prima passata completa, l'elemento massimo assoluto avrà necessariamente raggiunto l'ultima cella dell'array.
4. Si ripete la passata per i rimanenti $n - 1$ elementi, congelando progressivamente gli elementi massimi a destra.

L'Ottimizzazione del Flag scambiato

Nella sua versione base, il Bubble Sort esegue sempre tutti i cicli anche se l'array viene ordinato a metà dell'opera. Possiamo ottimizzarlo introducendo una variabile booleana `scambiato`: se durante una passata completa non avviene alcun scambio fisico, significa che l'array è già perfettamente ordinato ed è inutile continuare.



Figura 10.3: Bubble Sort in Azione

Codice C Completo del Bubble Sort Ottimizzato:

```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4  void swap(int *a, int *b) {
5      int temp = *a;
6      *a = *b;
7      *b = temp;
8  }
9
10 void bubble_sort(int arr[], int n) {
11     bool scambiato;
12     for (int i = 0; i < n - 1; i++) {
13         scambiato = false;
14
15         // Gli ultimi 'i' elementi sono già consolidati (congelati)
16         for (int j = 0; j < n - i - 1; j++) {
17             if (arr[j] > arr[j + 1]) {
18                 swap(&arr[j], &arr[j + 1]);
19                 scambiato = true; // Uno scambio ? avvenuto
20             }
21         }
22
23         // Se non ci sono stati scambi, l'array è ordinato. Uscita
24         ⇨ anticipata!
25         if (!scambiato) {
26             break;
27         }
28     }
29
30 int main(void) {
31     int dati[] = {5, 1, 4, 2, 3};
32     int n = 5;

```

```

33
34     bubble_sort(dati, n);
35
36     printf("Bubble Sort: ");
37     for(int i = 0; i < n; i++) {
38         printf("%d ", dati[i]);
39     }
40     printf("\n");
41     return 0;
42 }

```

10.3.3 Insertion Sort: “L’Ordinamento delle Carte”

L’**Insertion Sort** ricalca fedelmente il modo in cui ordiniamo le carte da gioco che teniamo in mano:

1. Consideriamo il primo elemento (`arr[0]`) come un sotto-array già ordinato di lunghezza 1.
2. Esaminiamo il secondo elemento. Lo “estriamo” memorizzandolo in una variabile temporanea `temp` (la nostra carta in mano).
3. **Fase di Slittamento (Shifting):** Confrontiamo `temp` a ritroso con gli elementi della porzione ordinata a sinistra. Finché questi elementi sono maggiori di `temp`, li facciamo slittare a destra di una posizione (sovrascrivendo la cella successiva).
4. **Fase di Inserimento:** Non appena troviamo un elemento minore o uguale a `temp` (o raggiungiamo l’inizio dell’array), inseriamo la nostra “carta” nella posizione vuota rimasta libera.

Nota

Nota A differenza del Selection/Bubble Sort che lavorano effettuando scambi diretti (swap), l’Insertion Sort lavora spostando i dati tramite slittamenti progressivi in un’unica direzione, riducendo il numero di accessi in scrittura sulla memoria RAM.

Codice C Completo dell’Insertion Sort:

```

1  #include <stdio.h>
2
3  void insertion_sort(int arr[], int n) {
4      // arr[0] ? assunto gi? ordinato. Partiamo dall'indice 1
5      for (int i = 1; i < n; i++) {
6          int temp = arr[i]; // Estraiamo l'elemento corrente
7          int j = i - 1;
8
9          // Spostiamo verso destra tutti gli elementi maggiori di temp
10         while (j >= 0 && arr[j] > temp) {
11             arr[j + 1] = arr[j]; // Slittamento fisico a destra

```



Figura 10.4: Insertion Sort in Azione

```

12         j--; // Continuiamo la scansione a ritroso
13     }
14
15     // Inseriamo temp nella posizione corretta liberata dallo
16     ↪ slittamento
17     arr[j + 1] = temp;
18 }
19
20 int main(void) {
21     int dati[] = {4, 2, 5, 1, 3};
22     int n = 5;
23
24     insertion_sort(dati, n);
25
26     printf("Insertion Sort: ");
27     for(int i = 0; i < n; i++) {
28         printf("%d ", dati[i]);
29     }
30     printf("\n");
31     return 0;
32 }

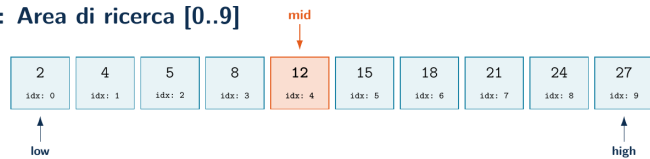
```

10.4 La Ricerca Dicotomica (Binary Search)

10.4.1 Ricerca Lineare vs Dicotomica

La ricerca di un elemento in un array può essere condotta in due modi:

Passo 1: Area di ricerca [0..9]



Condizione: $M[mid] (12) < Chiave (24)$
 Azione: L'elemento cercato si trova a destra.
 Spostiamo l'indice inferiore: $low = mid + 1 (5)$.

Passo 2: Area di ricerca [5..9]



Condizione: $M[mid] (21) < Chiave (24)$
 Azione: L'elemento si trova ancora a destra.
 Spostiamo l'indice inferiore: $low = mid + 1 (8)$.

Passo 3: Area di ricerca [8..9]



Condizione: $M[mid] (24) == Chiave (24)$
 Azione: Elemento trovato all'indice 8!
 Costo esecutivo: Soli 3 confronti anziché 10 per un array di queste dimensioni!

Figura 10.5: Ricerca Dicotomica in Azione

- **Ricerca Lineare:** Scorre l'array dall'inizio e confronta ogni cella con la chiave cercata. Funziona su array sia ordinati che disordinati, ma richiede in media $N/2$ confronti e N confronti nel caso peggiore.
- **Ricerca Dicotomica (Binary Search):** Richiede che l'array sia **rigorosamente ordinato**. Adotta una strategia di tipo *Divide et Impera*: confronta la chiave con l'elemento centrale. Se sono diversi, scarta la metà dell'array in cui la chiave non può risiedere e ripete la ricerca nella metà rimanente.

10.4.2 Prevenire l'Overflow nel Calcolo del Punto Medio

Nel calcolare l'indice centrale, la formula intuitiva sarebbe $(low + high) / 2$. Tuttavia, se low e $high$ assumono valori molto grandi (vicini al limite superiore del tipo `int`, pari a $2^{31} - 1$), la loro somma diretta provocherà un **integer overflow**, producendo un valore negativo e causando un crash di segmentazione durante l'accesso all'array.

Per evitare questo bug critico, si adotta la formula aritmeticamente equivalente ma sicura:

$$mid = low + \frac{high - low}{2}$$

In questo modo, la sottrazione $high - low$ non potrà mai superare la dimensione dell'area, scongiurando il rischio di overflow.

Codice C della Ricerca Dicotomica:

```
1 #include <stdio.h>
2
3 // Restituisce l'indice dell'elemento se trovato, altrimenti -1
4 int ricerca_dicotomica(int arr[], int n, int chiave) {
5     int low = 0;
6     int high = n - 1;
7 }
```

```

8   while (low <= high) {
9       // Calcolo sicuro del punto medio per evitare overflow numerico
10      int mid = low + (high - low) / 2;
11
12      if (arr[mid] == chiave) {
13          return mid; // Trovato! Restituiamo l'indice
14      }
15
16      if (arr[mid] < chiave) {
17          low = mid + 1; // La chiave si trova nella met? destra
18      } else {
19          high = mid - 1; // La chiave si trova nella met? sinistra
20      }
21  }
22
23  return -1; // Chiave non presente nell'array
24 }
25
26 int main(void) {
27     int elenco[] = {2, 4, 5, 8, 12, 15, 18, 21, 24, 27};
28     int n = 10;
29     int chiave = 24;
30
31     int pos = ricerca_dicotomica(elenco, n, chiave);
32
33     if (pos != -1) {
34         printf("Trovato! Il valore %d si trova all'indice: %d\n", chiave,
35             ↪ pos);
36     } else {
37         printf("Valore %d non trovato.\n", chiave);
38     }
39     return 0;
40 }

```

10.4.3 Analisi quantitativa dell'efficienza

Vediamo numericamente perché la ricerca dicotomica è una svolta ingegneristica per la CPU. Supponiamo di cercare una chiave in un array ordinato contenente esattamente $N = 1.048.576$ elementi (circa 1 milione).

- Con la **Ricerca Lineare**, nel caso peggiore la CPU eseguirà **1.048.576 confronti**.
- Con la **Ricerca Dicotomica**, la dimensione dell'area si dimezza ad ogni passo:

$$1.048.576 \rightarrow 524.288 \rightarrow 262.144 \rightarrow 131.072 \rightarrow \dots \rightarrow 1$$

Poiché $2^{20} = 1.048.576$, la ricerca si concluderà in al massimo **20 confronti**.

Passare da 1 milione di operazioni a solo 20 riduce drasticamente l'utilizzo di energia, tempo e risorse di calcolo della CPU, rendendo la ricerca dicotomica incredibilmente efficiente su grandi moli di dati.

10.5 Quiz di Auto-valutazione

Quiz 1

Domanda

1. Qual è la condizione essenziale per poter applicare la ricerca dicotomica (Binary Search) su un array?

- A) L'array deve essere allocato dinamicamente nello Heap.
- B) L'array deve essere preventivamente ordinato.
- C) L'array deve contenere solo numeri positivi.
- D) L'array non deve contenere duplicati.

Spiegazione: La ricerca dicotomica si basa sulla certezza matematica che, confrontando la chiave con l'elemento centrale mid , se $arr[mid] < chiave$, allora la chiave non può risiedere nella metà sinistra. Questo principio crolla se gli elementi non sono ordinati.

Risposta corretta: B

Soluzione

Quiz 2

Domanda

2. Quale delle seguenti formule previene il bug di overflow aritmetico nel calcolo del punto medio?

- A) $(low + high) / 2$
- B) $low + (high - low) / 2$
- C) $high - (high - low) / 2$
- D) $(low / 2) + (high / 2)$

Spiegazione: Se low e $high$ sono molto grandi, la loro somma $low + high$ può superare il massimo numero intero rappresentabile ($2^{31} - 1$), andando in overflow (diventando negativa). La sottrazione $high - low$ è invece sempre minore del massimo indice ed è quindi sicura.

Risposta corretta: B

Soluzione

Quiz 3

Domanda

3. Nel Selection Sort, quante operazioni fisiche di swap vengono eseguite al massimo in un array di N elementi?

- A) N^2
- B) $N \log N$
- C) $N - 1$
- D) Nessuna, non usa swap.

Soluzione
Risposta corretta: C
Spiegazione: Il Selection Sort effettua al massimo uno swap per ogni iterazione del ciclo esterno. Poiché il ciclo esterno viene eseguito $N - 1$ volte, il numero massimo di swap eseguiti è esattamente $N - 1$. Questa proprietà lo rende ideale quando scrivere in memoria è molto costoso.

Quiz 4

Domanda

4. Che cosa succede nel Bubble Sort ottimizzato se durante una passata completa dell'array non viene eseguito nessuno swap?

- A) L'algoritmo va in loop infinito.
- B) Il flag scambiato rimane `false` e l'algoritmo termina immediatamente.
- C) L'algoritmo ricomincia dall'inizio azzerando gli indici.
- D) Viene generato un errore a runtime.

Soluzione
Risposta corretta: B
Spiegazione: Se non avviene alcuno scambio, significa che ogni elemento dell'array è già minore o uguale al suo successore. Il flag scambiato (inizializzato a `false`) non viene alterato, e la condizione `if ( !scambiato )` forza l'uscita anticipata dal ciclo (break).

Quiz 5

Domanda

5. Qual è la caratteristica distintiva del meccanismo di Insertion Sort rispetto a Bubble e Selection Sort?

- A) Non esegue confronti tra elementi.
- B) Sposta gli elementi tramite slittamento progressivo (shifting) anziché swap continui.
- C) Funziona solo su stringhe e non su interi.
- D) Richiede un array ausiliario per copiare i dati.

Spiegazione: L'insertion Sort estrae temporaneamente l'elemento corrente in tempo e fa slittare a destra di una posizione tutti gli elementi ordinati maggiori di esso. Non esegue scambi bidirezionali continui (swap), riducendo le scritture in RAM.

Risposta corretta: B

Soluzione

Quiz 6

Domanda

6. Se cerchiamo un elemento inesistente in un array ordinato di 4096 elementi con la ricerca dicotomica, quanti confronti faremo al massimo?

- A) 4096
- B) 2048
- C) 12
- D) 7

Spiegazione: Poiché ad ogni passo l'area di ricerca viene divisa per due, cerchiamo il più piccolo esponente k tale che $2^k \geq 4096$. Dato che $2^{12} = 4096$, l'algoritmo eseguirà al massimo 12 confronti.

Risposta corretta: C

Soluzione

Quiz 7

Domanda

7. Qual è lo scenario peggiore (worst-case) per il Bubble Sort non ottimizzato?

- A) Array già ordinato in senso crescente.
- B) Array ordinato in senso decrescente (inverso).
- C) Array contenente solo elementi identici.
- D) Array vuoto.

Spiegazione: Se l'array è ordinato al contrario, ogni singolo confronto richiederà uno swap per far risalire l'elemento adiacente. L'algoritmo dovrà effettuare il numero massimo di scambi fisici possibili.

Risposta corretta: B

Soluzione

Quiz 8

Domanda

8. Nella funzione `swap(int *a, int *b)`, perché è necessario passare gli indirizzi delle variabili (puntatori)?

- A) Perché il C passa i parametri per valore (copia), quindi senza puntatori modificherebbero solo copie locali delle variabili.
- B) Perché i puntatori rendono l'esecuzione più lenta ma sicura.
- C) Perché lo Stack Frame non permette di scambiare variabili locali.
- D) Perché è imposto dalla sintassi delle funzioni di ordinamento.

Soluzione: Il passaggio dei parametri in C avviene per valore. Se passassimo direttamente `swap(x, y)`, la funzione scambierebbe i valori all'interno dei suoi parametri locali `a` e `b`, lasciando del tutto inalterate le variabili originali `x` e `y` del chiamante.

Risposta corretta: A

Soluzione

Quiz 9

Domanda

9. Quale algoritmo tra Selection, Bubble e Insertion Sort si adatta meglio ad ordinare elementi che vengono inseriti in tempo reale uno alla volta?

- A) Selection Sort, perché cerca il minimo.
- B) Bubble Sort, perché fa risalire le bollicine.
- C) Insertion Sort, perché inserisce l'elemento al posto giusto slittando solo il necessario.
- D) Nessuno di essi.

Soluzione: L'Insertion Sort assume che la porzione a sinistra sia già ordinata. Se arriva un nuovo elemento in coda, basta farlo slittare all'indietro fino alla sua corretta collocazione. È l'approccio ideale per l'ordinamento "in-place" online.

Risposta corretta: C

Soluzione

Quiz 10

Domanda

10. Che cosa restituisce la funzione `ricerca\{}_dicotomica` se la chiave cercata è presente più volte nell'array ordinato?

- A) L'indice della prima occorrenza in assoluto.

- B) L'indice dell'ultima occorrenza.
- C) Un indice arbitrario tra le occorrenze della chiave (dipendente da dove si ferma `mid`).
- D) Ritorna `-1` per errore.

Soluzione
Risposta corretta: C
Spiegazione: La Binary Search standard si ferma non appena trova `arr[mid] == chiave`. Se ci sono duplicati, l'indice restituito dipenderà da quale occorrenza si trova ad essere colpita per prima dal punto medio `mid`. Per trovare la prima o l'ultima occorrenza servono varianti specifiche dell'algoritmo.

10.6 Esercizi Pratici

Esercizio 1: Trova l'Indice del Minimo

Domanda

Scrivete una funzione `C int trova_minimo(int arr[], int start, int end)` che restituisca l'indice dell'elemento minimo presente nell'array tra le posizioni `start` ed `end` (inclusi).

Spiegazione dell'algoritmo: Inizializziamo l'indice del minimo con il valore di partenza start. Scansioniamo l'array con un ciclo da start + 1 a end. Se troviamo un valore inferiore a quello puntato da idx\{\}_min, aggiorniamo idx\{\}_min. Al termine restituiamo l'indice.

```

1  #include <stdio.h>
2
3  int trova_minimo(int arr[], int start, int end) {
4      int idx_min = start;
5      for (int i = start + 1; i <= end; i++) {
6          if (arr[i] < arr[idx_min]) {
7              idx_min = i; // Aggiorna l'indice del minimo
8          }
9      }
10     return idx_min;
11 }
12
13 int main(void) {
14     int vet[] = {9, 4, 2, 7, 5, 1, 8};
15     // Cerca il minimo tra l'indice 1 e l'indice 5 (valori da 4 a
16     int pos = trova_minimo(vet, 1, 5);
17     printf("Il minimo tra gli indici 1 e 5 si trova all'indice %d (
18     valore: %d)\n", pos, vet[pos]);
19     return 0;

```

Soluzione

Esercizio 2: Bubble Sort Decrescente

Domanda

Modificate l'algoritmo del Bubble Sort per ordinare l'array in ordine decrescente (dal più grande al più piccolo).

Spiegazione dell'algoritmo: Per invertire l'ordine dell'ordinamento è sufficiente invertire la condizione del confronto all'interno del ciclo interno: modificando `arr[j] < arr[j + 1]` in `arr[j] > arr[j + 1]`, facciamo risalire a destra i valori più piccoli, spingendo a sinistra i valori più grandi.

```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4  void swap(int *a, int *b) {
5      int temp = *a;
6      *a = *b;
7      *b = temp;
8  }
9
10 void bubble_sort_descendente(int arr[], int n) {
11     bool scambiato;
12     for (int i = 0; i < n - 1; i++) {
13         scambiato = false;
14         for (int j = 0; j < n - i - 1; j++) {
15             // Per ordinare in modo decrescente, scambiamo se l'
16             // elemento di sinistra è MINORE di quello a destra
17             if (arr[j] < arr[j + 1]) {
18                 swap(&arr[j], &arr[j + 1]);
19                 scambiato = true;
20             }
21             if (!scambiato) break;
22         }
23     }
24
25     int vett[] = {3, 1, 4, 5, 2};
26     bubble_sort_descendente(vett, 5);
27     printf("Decrescente: ");
28     for (int i = 0; i < 5; i++) {
29         printf("%d ", vett[i]);
30     }
31     printf("\n");
32     return 0;
33 }
34

```

Soluzione

Esercizio 3: Conta Confronti e Scambi nel Selection Sort

Domanda

Implementate una versione del Selection Sort che tenga traccia del numero totale di confronti effettuati tra elementi e del numero di swap eseguiti, stampando i risultati alla fine.

Spiegazione dell'algoritmo: Incrementiamo la variabile confronti ad ogni esecuzione della condizione `if` del ciclo interno e incrementiamo scambi solo all'interno del blocco condizionale `if (min_idx != i)` che esegue lo swap. Su 5 elementi disordinati vedremo 10 confronti e al massimo 4 swap.

```

1  #include <stdio.h>
2
3  void selection_sort_tracciato(int arr[], int n) {
4      int confronti = 0;
5      int scambi = 0;
6
7      for (int i = 0; i < n - 1; i++) {
8          int min_idx = i;
9          for (int j = i + 1; j < n; j++) {
10             confronti++;
11             if (arr[j] < arr[min_idx]) {
12                 min_idx = j;
13             }
14         }
15         if (min_idx != i) {
16             int temp = arr[i];
17             arr[i] = arr[min_idx];
18             arr[min_idx] = temp;
19             scambi++;
20         }
21         printf("Confronti eseguiti: %d\n", confronti);
22         printf("Swap eseguiti: %d\n", scambi);
23     }
24 }
25
26 int main(void) {
27     int vet[] = {4, 2, 5, 1, 3};
28     selection_sort_tracciato(vet, 5);
29     return 0;
30 }

```

Soluzione

Esercizio 4: Bubble Sort Bidirezionale (Cocktail Shaker)

Domanda

Sort)

Scrivete una variante del Bubble Sort che esegua passate alternate: una da sinistra verso destra (portando il massimo a destra) e una da destra verso sinistra (portando il minimo a sinistra).



Spiegazione dell'algoritmo: Chiamato anche *Cocktail Shaker Sort*, questo approccio ottimizza il Bubble Sort riducendo il problema degli elementi piccoli situati in fondo a destra (chiamati "tartarughe", che impiegherebbero molto tempo a risalire a sinistra in passate unidirezionali).

```

1  #include <csdlib.h>
2
3  void swap(int *a, int *b) {
4      int temp = *a;
5      *a = *b;
6      *b = temp;
7  }
8
9
10 void cocktail_sort(int arr[], int n) {
11     bool scambiato = true;
12     int start = 0;
13     int end = n - 1;
14
15     while (scambiato) {
16         scambiato = false;
17
18         // Passata da sinistra a destra (Bubble Sort classico)
19         for (int i = start; i < end; i++) {
20             if (arr[i] > arr[i + 1]) {
21                 swap(&arr[i], &arr[i + 1]);
22                 scambiato = true;
23             }
24         }
25
26         // Se non ci sono stati scambi, l'array è ordinato
27         if (!scambiato) break;
28
29         // Riduciamo end perché l'elemento massimo è congelato a
30         // destra
31         end--;
32
33         scambiato = false;
34
35         // Passata da destra a sinistra
36         for (int i = end - 1; i >= start; i--) {
37             if (arr[i] > arr[i + 1]) {
38                 swap(&arr[i], &arr[i + 1]);
39                 scambiato = true;
40             }
41         }
42
43         // Incrementiamo start perché il minimo è congelato a
44         // sinistra
45         start++;
46     }
47
48     int vett[] = {5, 1, 4, 2, 3};
49     cocktail_sort(vett, 5);
50     printf("Ordinato: ");
51     for (int i = 0; i < 5; i++) {
52         printf("%d ", vett[i]);
53     }
54     printf("\n");
55     return 0;
56 }

```


Esercizio 5: Ricerca Lineare di Tutte le Occorrenze**Domanda**

Scrivete una funzione `int cerca_tutte(int arr[], int n, int chiave, int indici[])` che cerchi una chiave in un array disordinato e salvi tutti gli indici in cui compare all'interno dell'array `indici`. La funzione deve restituire il numero totale di occorrenze trovate.

Spiegazione dell'algoritmo: Scorriamo l'array cella per cella. Quando `arr[i]` è uguale alla chiave, salviamo il valore dell'indice `i` nella posizione `contatore` dell'array di output `indici` e incrementiamo `contatore`.

```

1  #include <stdio.h>
2
3  int cerca_tutte(int arr[], int n, int chiave, int indici[]) {
4      int contatore = 0;
5      for (int i = 0; i < n; i++) {
6          if (arr[i] == chiave) {
7              indici[contatore] = i; // Salva l'indice corrente
8              contatore++;
9          }
10         return contatore; // Restituisce il numero totale
11     }
12 }
13
14 int main(void) {
15     int vett[] = {4, 2, 5, 2, 3, 2, 8};
16     int ris[7]; // Vettore per salvare gli indici trovati
17     int occorrenze = cerca_tutte(vett, 7, 2, ris);
18     printf("Trovate %d occorrenze del valore 2 agli indici: ",
19         occorrenze);
20     for (int i = 0; i < occorrenze; i++) {
21         printf("%d ", ris[i]);
22     }
23     printf("\n");
24     return 0;
25 }

```

Soluzione**Esercizio 6: Inserimento Ordinato****Domanda**

Scrivete una funzione `void inserimento_ordinato(int arr[], int *n, int max_size, int valore)` che inserisca un nuovo valore all'interno di un array già ordinato di dimensione logica `*n`, spostando gli elementi necessari per preservare l'ordinamento. La dimensione logica dell'array deve essere incrementata tramite il puntatore.

Spiegazione dell'algoritmo: Questo algoritmo è una singola passata del ciclo interno dell'insertion Sort. Partiamo dalla fine dell'array ($*n - 1$) e facciamo slittare a destra gli elementi maggiori del nuovo valore, inserendolo poi nello spazio vuoto.

```

1  #include <stdio.h>
2
3  void inserimento_ordinato(int arr[], int *n, int max_size, int
4      ↪ valore) {
5      if (*n >= max_size) {
6          printf("Errore: array pieno!\n");
7          return;
8      }
9      int j = *n - 1;
10     // Slitta gli elementi a destra finché sono maggiori del valore
11     ↪ da inserire
12     while (j >= 0 && arr[j] > valore) {
13         arr[j + 1] = arr[j];
14         j--;
15     }
16     // Inserisce il valore nella posizione liberata
17     arr[j + 1] = valore;
18     // Incrementa la dimensione logica puntata
19     (*n)++;
20 }
21
22 int main(void) {
23     int vet[10] = {2, 4, 7, 8, 12};
24     int n = 5; // Dimensione logica iniziale
25     inserimento_ordinato(vet, &n, 10, 5);
26     printf("Array dopo inserimento ordinato di 5: ");
27     for (int i = 0; i < n; i++) {
28         printf("%d ", vet[i]);
29     }
30     printf("\n");
31     return 0;
32 }
33
34
35

```

Soluzione

Esercizio 7: Ricerca Dicotomica Ricorsiva

Domanda

Riscrivete l'algoritmo della ricerca dicotomica in modalità ricorsiva sfruttando i parametri di frontiera low e high.

Spiegazione dell'algoritmo: Se $low > high$, l'elemento non è presente (caso base di fallimento). Altrimenti calcoliamo mid e confrontiamo. Se $arr[mid]$ è maggiore della chiave, richiamiamo ricorsivamente la funzione restringendo l'intervallo a $[low, mid - 1]$, altrimenti lo restringiamo a $[mid + 1, high]$.

```

1  #include <stdio.h>
2
3  int ricerca_dicotomica_ricorsiva(int arr[], int low, int high, int
4      // Caso base: area di ricerca esaurita
5      if (low > high) {
6          return -1;
7      }
8
9      int mid = low + (high - low) / 2;
10
11     if (arr[mid] == chiave) {
12         return mid; // Trovato!
13     }
14
15     if (arr[mid] > chiave) {
16         // Cerca nella metà sinistra
17         return ricerca_dicotomica_ricorsiva(arr, low, mid - 1,
18             chiave);
19     } else {
20         // Cerca nella metà destra
21         return ricerca_dicotomica_ricorsiva(arr, mid + 1, high,
22             chiave);
23     }
24 }
25
26 int elenco[] = {2, 4, 5, 8, 12, 15, 18, 21, 24, 27};
27 int pos = ricerca_dicotomica_ricorsiva(elenco, 0, 9, 24);
28 printf("Indice trovato ricorsivamente: %d\n", pos);
29 return 0;

```

Soluzione

Esercizio 8: Selection Sort su Array di Float

Domanda

Adattate la funzione del Selection Sort per ordinare un array di numeri reali in virgola mobile (float).

Spiegazione dell'algoritmo: L'algoritmo rimane strutturalmente identico. È stato sufficiente modificare la dichiarazione del tipo del parametro formale dell'array (float arr[]) e della variabile temporanea all'interno dello swap (float temp).

```

1  #include <stdio.h>
2
3  void selection_sort_float(float arr[], int n) {
4      for (int i = 0; i < n - 1; i++) {
5          int min_idx = i;
6          for (int j = i + 1; j < n; j++) {
7              if (arr[j] < arr[min_idx]) {
8                  min_idx = j;
9              }
10             if (min_idx != i) {
11                 float temp = arr[i];
12                 arr[i] = arr[min_idx];
13                 arr[min_idx] = temp;
14             }
15         }
16     }
17 }
18
19 int main(void) {
20     float vet[] = {3.14, 1.41, 2.71, 0.707, 1.618};
21     selection_sort_float(vet, 5);
22
23     printf("Float ordinati: ");
24     for (int i = 0; i < 5; i++) {
25         printf("%.3f ", vet[i]);
26     }
27     printf("\n");
28     return 0;
29 }

```

Soluzione

Esercizio 9: Rimozione dei Duplicati in un Array Ordinato

Domanda

Scrivete una funzione `int rimuovi\_duplicati(int arr[], int n)` che prenda un array già ordinato e rimuova gli elementi duplicati in situ (senza vettori ausiliari). La funzione deve restituire la nuova dimensione logica dell'array.

Spiegazione dell'algoritmo: Sfruttiamo l'ordinamento: gli elementi uguali sono adiacenti. Scorriamo il vettore confrontando `arr[i]` con `arr[i+1]`. Se sono diversi, spostiamo `arr[i]` all'indice `j` (che tiene traccia della porzione senza duplicati) e avanziamo `j`.

```

1  #include <stdio.h>
2
3  int rimuovi_duplicati(int arr[], int n) {
4      if (n == 0 || n == 1) return n;
5
6      int j = 0; // Indice per memorizzare gli elementi unici
7      for (int i = 0; i < n - 1; i++) {
8          // Se l'elemento corrente è diverso dal successivo, lo
9          // salviamo
10         if (arr[i] != arr[i + 1]) {
11             arr[j] = arr[i];
12             j++;
13         }
14         // Salva l'ultimo elemento che non è stato confrontato nel
15         // ciclo
16         arr[j] = arr[n - 1];
17         j++;
18     }
19     return j; // Nuova dimensione logica
20 }
21
22 int main(void) {
23     int dati[] = {1, 1, 2, 2, 3, 4, 4, 5};
24     int n = 8;
25     int nuova_dim = rimuovi_duplicati(dati, n);
26     printf("Array senza duplicati (%d elementi): ", nuova_dim);
27     for (int i = 0; i < nuova_dim; i++) {
28         printf("%d ", dati[i]);
29     }
30     printf("\n");
31     return 0;
32 }
```

Soluzione

Esercizio 10: Ricerca Dicotomica con Verifica Indici

Domanda

Limite

Scrivete una variante di `ricerca_dicotomica` che, se la chiave è presente più volte, garantisca di restituire l'indice della **prima occorrenza** (quella più a sinistra) all'interno dell'array.

Soluzione

```

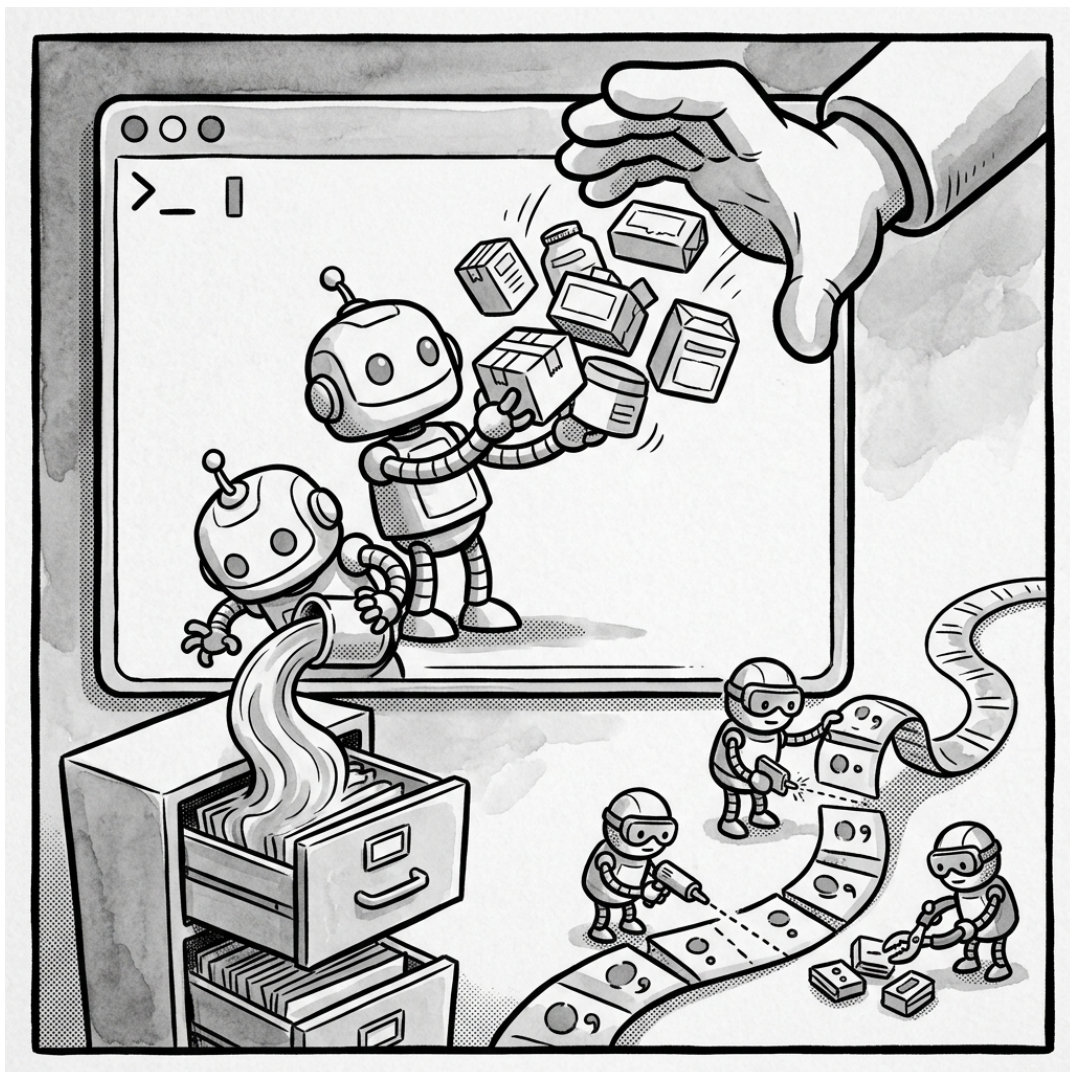
1  #include <stdio.h>
2
3  int prima-occorrenza-dicotomica(int arr[], int n, int chiave) {
4      int low = 0;
5      int high = n - 1;
6      int ris = -1; // Memorizza la posizione temporanea
7
8      while (low <= high) {
9          int mid = low + (high - low) / 2;
10
11         if (arr[mid] == chiave) {
12             ris = mid; // Trovata! Memorizziamo l'indice
13             high = mid - 1; // Continuiamo a cercare a SINISTRA per
14             // trovare occorrenze precedenti
15         } else if (arr[mid] < chiave) {
16             low = mid + 1;
17         } else {
18             high = mid - 1;
19         }
20     }
21     return ris;
22 }
23
24 int main(void) {
25     // Array ordinato con duplicati del valore 12
26     int elenco[] = {2, 4, 12, 12, 12, 15, 18, 21};
27     int pos = prima-occorrenza-dicotomica(elenco, 8, 12);
28     printf("La prima occorrenza di 12 si trova all'indice: %d\n",
29         pos); // Deve stampare 2
30     return 0;
31 }

```

Spiegazione dell'algoritmo: Quando troviamo `arr[mid] == chiave`, anziché fermarci restituendo subito `mid`, salviamo questo indice in una variabile di appoggio `ris` e forziamo la ricerca a procedere nella metà sinistra restringendo il range di ricerca con `high = mid - 1`. Se non ci sono altre occorrenze, la ricerca terminerà restituendo l'ultima salvata in `ris`.

LEZIONE 11

Interazione con l'OS: CLI, Flussi di Testo e Tokenizzazione



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 11

Interazione con l'OS: CLI, Flussi di Testo e Tokenizzazione

Fino ad oggi, i nostri programmi hanno interagito con l'utente in modo esclusivamente interattivo, fermandosi ad attendere dati digitati da tastiera a run-time. Tuttavia, nello sviluppo software professionale e nell'automazione di sistema, i programmi devono poter ricevere parametri direttamente all'avvio dal terminale (Command Line Interface) ed elaborare file in modo robusto senza bloccarsi davanti a dati corrotti.

In questa lezione vedremo come il Sistema Operativo passa parametri al processo (l'anatomia di `argc` e `argv` in memoria), come differenziare i dati dall'output diagnostico usando lo **Standard Error (`stderr`)**, come implementare letture stabili prevenendo le trappole intrinseche di `scanf` e, infine, come effettuare il parsing robusto e la tokenizzazione in memoria RAM tramite `sscanf` e `strtok`.

11.1 La Riga di Comando e lo Standard Error (CLI & STDERR)

11.1.1 Il Limite dell'Isolamento interattivo

I comandi di sistema professionali (come `cp`, `ls` o `git`) non avviano dialoghi interattivi con domande del tipo *"Inserisci la sorgente:"* o *"Inserisci la destinazione:"*. Essi ricevono le istruzioni operative direttamente al momento del lancio, ad esempio scrivendo `cp foto.png backup/`. Questo modello consente di concatenare ed automatizzare i comandi all'interno di script. In C, questa interazione viene gestita tramite due parametri speciali passati alla funzione `main`.

11.1.2 L'Anatomia di `argc` e `argv` nello Stack

Per accogliere i parametri di lancio, definiamo il `main` con la seguente firma:

```
1 int main(int argc, char *argv[])
```

- **`argc` (Argument Count):** Un intero che conta il numero totale di argomenti digitati sul terminale, incluso il nome stesso del programma. È sempre ≥ 1 .

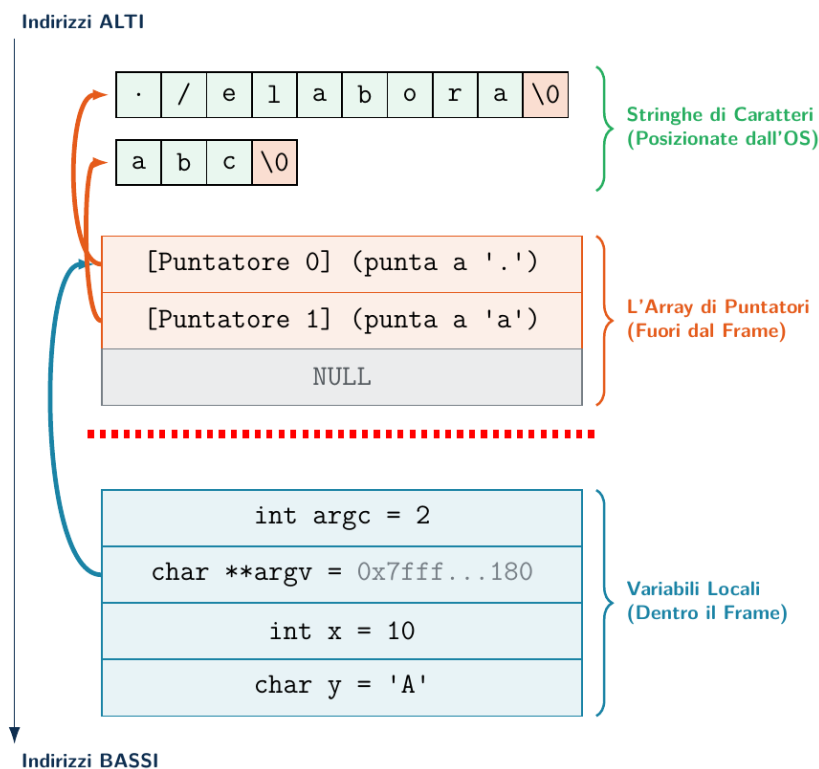


Figura 11.1: Anatomia di argc e argv nello Stack

- **argv (Argument Vector):** Un array di puntatori a caratteri (stringhe).
 - argv[0] contiene sempre la stringa con il nome dell'eseguibile (ad es. `"/.elabora"`).
 - argv[1], argv[2], ..., argv[argc-1] contengono i parametri di lancio veri e propri.
 - argv[argc] è un puntatore nullo (NULL), che serve come sentinella di fine array.

La mutabilità di argv

Un errore comune è ritenere che le stringhe lette da argv si trovino nell'area di memoria a sola lettura (RoData). In realtà, all'avvio del processo, il *loader* del Sistema Operativo deposita i testi degli argomenti **all'inizio assoluto della memoria Stack** (agli indirizzi fisici più alti), seguiti dall'array di puntatori argv. Solo dopo viene allocato lo Stack Frame della funzione main.

Essendo allocate nello Stack, le stringhe di input sono **mutabili**: il programma può scriverci dentro e alterarle (es. `argv[1][0] = 'X'`) senza provocare crash di segmentazione.

Codice C: Stampare e modificare gli argomenti a riga di comando

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, char *argv[]) {
5     printf("Hai lanciato il programma con %d argomenti.\n", argc);
6
7     for (int i = 0; i < argc; i++) {
8         printf("argv[%d]: \"%s\"\n", i, argv[i]);
9     }
10
11     // Intercettare un'opzione di aiuto
12     if (argc > 1 && strcmp(argv[1], "--help") == 0) {
13         printf("\nUso: ./programma [valore_1] [operatore] [valore_2]\n");
14     }
15
16     // Dimostrazione di mutabilit  nello Stack
17     if (argc > 1) {
18         argv[1][0] = 'X';
19         printf("Argomento 1 modificato fisicamente: %s\n", argv[1]);
20     }
21
22     return 0;
23 }
```

11.1.3 Lo Standard Error (stderr)

Se un'applicazione stampa un messaggio di errore (come *"Impossibile aprire il file"*) usando una normale `printf`, questo messaggio finisce sullo **Standard Output (stdout)** insieme ai dati regolari prodotti dal programma. Ci  rende difficile separare i dati puliti dai diagnostici d'errore.

Per ovviare a questo problema, ogni programma C dispone fin dall'avvio di tre flussi standard (identificati a basso livello da numeri detti *File Descriptor*):

1. **stdin (File Descriptor 0)**: Canale di ingresso dei dati (tastiera).
2. **stdout (File Descriptor 1)**: Canale di uscita standard dei dati (schermo del terminale).
3. **stderr (File Descriptor 2)**: Canale di uscita indipendente dedicato a errori e messaggi diagnostici.

Utilizzando la funzione `fprintf` (dove la *f* sta per file/stream) possiamo deviare i messaggi sul canale desiderato:

```
1 fprintf(stdout, "Questo e' un dato normale.\n"); // Equivalente a printf
   ⇨ (...)
2 fprintf(stderr, "ERRORE CRITICO: Parametro non valido!\n"); // Va su
   ⇨ stderr
```

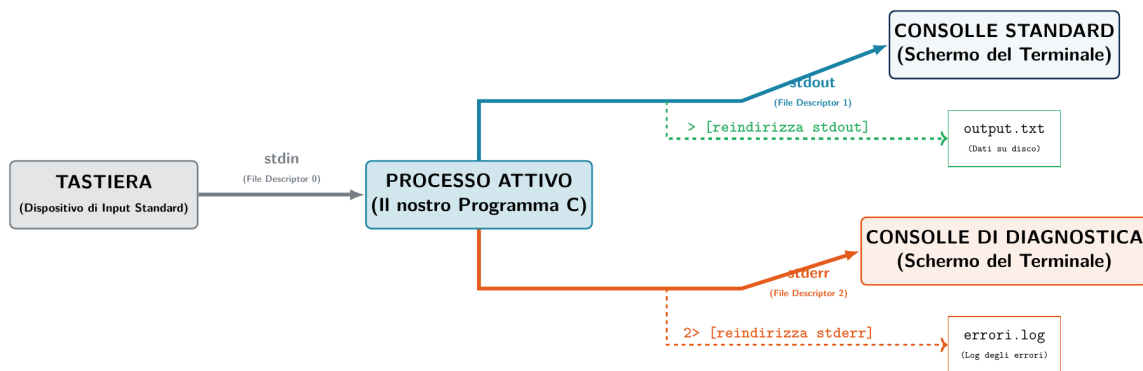


Figura 11.2: Architettura dei Flussi Standard Unix

Reindirizzamento dei Flussi da Terminale

L'esistenza di canali separati permette al terminale di effettuare operazioni di redirezione:

- **Devviare solo stdout su file:** `make > output.txt` (lo schermo rimarrà pulito, i dati andranno nel file, ma gli eventuali errori verranno comunque stampati a schermo).
- **Devviare solo stderr su file:** `./programma 2> errori.log` (i dati corretti vanno a schermo, gli errori vengono isolati nel log).
- **Devviare entrambi su file separati:** `./programma > output.txt 2> errori.log`
- **Unire stderr a stdout su unico file:** `./programma > tutto.txt 2>\{\}&1` (devia il canale 2 nel canale 1).

11.2 I Limiti dell'Input e l'Acquisizione Robusta

11.2.1 Le insidie nascoste di scanf

La funzione `scanf` è pensata per l'acquisizione formattata rapida, ma presenta gravi limiti di stabilità in scenari reali:

1. **Il problema del tipo errato:** Se il codice richiede un numero (`scanf("\{\}%d", \{\}&eta)`) e l'utente inserisce la lettera 'A', la funzione fallisce la conversione e restituisce immediatamente il controllo al programma senza valorizzare la variabile. Se quest'ultima non è inizializzata, conterrà dati casuali dello stack ("spazzatura"). Inoltre, la 'A' rimarrà incastrata nel buffer di sistema, mandando in blocco o in loop infinito le successive chiamate di lettura.
2. **Il limite degli spazi:** La lettura di stringhe tramite `scanf("\{\}%s", buffer)` si interrompe sistematicamente davanti al primo carattere di spaziatura (spazio, tabulazione, invio). Non è quindi possibile acquisire stringhe contenenti frasi complete.

11.2.2 Tecniche di Lettura Sicura

Per garantire la stabilità del software, si applicano due regole fondamentali:

Regola 1: Verificare sempre il valore di ritorno di `scanf`

La funzione `scanf` restituisce il numero di elementi convertiti con successo. Dobbiamo sempre verificarne il valore:

```
1 int eta = 0;
2 printf("Inserisci la tua eta': ");
3 if (scanf("%d", &eta) != 1) {
4     fprintf(stderr, "Errore: Inserire un valore numerico valido!\n");
5     // Gestione dell'errore e svuotamento del buffer...
6 }
```

Regola 2: Usare `fgets` per le stringhe

Per acquisire testi completi, si deve usare `fgets(buffer, dimensione, stream)`, che legge un'intera riga (inclusi gli spazi intermedi) e previene attacchi di *Buffer Overflow* rispettando rigorosamente la dimensione massima specificata:

```
1 char nome[50];
2 printf("Inserisci nome e cognome: ");
3 fgets(nome, sizeof(nome), stdin); // Legge in sicurezza fino a 49
   ↪ caratteri + '\0'
```

11.2.3 EOF, `getchar` e la lettura carattere per carattere

Per avere il controllo completo sull'input o svuotare il buffer, si ricorre alla lettura a basso livello tramite `getchar()`.

Perché `getchar()` restituisce `int` e non `char`?

`getchar()` estrae un singolo byte dallo stream di ingresso. Essa deve poter restituire tutti i 256 possibili valori di un carattere (da 0 a 255), più una condizione speciale che indichi la fine dell'input o del file: la costante **EOF (End Of File)**, definita solitamente come `-1`.

Se `getchar()` restituisse un tipo `char`, il comportamento varierebbe a seconda del compilatore, poiché lo standard C non specifica se il `char` base sia `signed` o `unsigned`:

- **Se `char` è `signed`:** il byte `0xFF` (valore ASCII 255) verrebbe letto come `-1`, venendo scambiato erroneamente per il segnale di EOF.
- **Se `char` è `unsigned`:** il segnale di EOF (`-1`) verrebbe convertito automaticamente in 255, venendo scambiato per un carattere valido e causando un ciclo infinito.

L'uso di un tipo `int` permette di mappare in sicurezza EOF al di fuori dell'intervallo dei caratteri validi, mantenendo distinti tutti i 256 valori positivi dal segnale di fine file.

Codice C: Costruire stringhe carattere per carattere invertendole

```

1  #include <stdio.h>
2
3  int main(void) {
4      char stringa[50];
5      printf("Inserisci la stringa (max 49 car. - premi invio per
   ↳ confermare): ");
6
7      int len = 0;
8      int c;
9
10     // Lettura manuale a basso livello con getchar()
11     do {
12         c = getchar();
13         // Escludiamo il newline e l'EOF, rispettando il limite fisico
   ↳ del buffer
14         if (c != '\n' && c != EOF && len < 49) {
15             stringa[len] = (char)c;
16             len++;
17         }
18     } while (c != EOF && c != '\n' && len < 49);
19
20     // Inserimento manuale del terminatore di stringa
21     stringa[len] = '\0';
22
23     printf("\nStringa invertita: ");
24     for (int i = len - 1; i >= 0; i--) {
25         printf("%c", stringa[i]);
26     }
27     printf("\n");
28
29     return 0;
30 }

```

11.3 Persistenza, File Stream e Tokenizzazione in Memoria

11.3.1 Apertura e Chiusura dei File: il tipo FILE

La persistenza delle informazioni viene gestita tramite puntatori a descrittori di file (`FILE *`). La struttura `FILE` racchiude i dettagli di stato del file gestiti dal Sistema Operativo (posizione corrente del cursore, buffer, flag di errore).

- **fopen("nome", "modo")**: Apre un file. I modi principali sono "r" (lettura, restituisce NULL se il file è assente), "w" (scrittura, crea un file vuoto cancellando l'eventuale contenuto preesistente) e "a" (append, scrive in coda senza cancellare).
- **fclose(fp)**: Chiude lo stream. Questa chiamata è fondamentale per forzare lo svuotamento dei buffer in memoria e garantire che i dati vengano fisicamente scritti sul disco prima che il programma termini.

11.3.2 Il Pericolo di `fscanf` nei File Reali

Cercare di interpretare un file di dati strutturati (es. un elenco CSV) tramite istruzioni dirette quali `fscanf(fp, "\\{}%s \\{}%d", ...)` è rischioso. Se una singola riga del file risulta malformata o contiene una stringa al posto di un numero (ad es. `Analisi\\{}_1 9 ASSENTE` invece di `Analisi\\{}_1 9 27`), la conversione fallisce, i dati incongruenti rimangono memorizzati nel buffer del file stream e il programma entrerà in loop infinito o salterà erroneamente tutte le righe rimanenti del file.

11.3.3 Robust Line-by-Line Parsing

La soluzione ingegneristica standard consiste nel dividere il processo di lettura dal processo di decodifica:

1. Si acquisisce l'intera riga "grezza" come stringa indipendente in memoria RAM usando la sicura `fgets()`.
2. Si effettua il parsing e la decodifica (tokenizzazione) della sola stringa in memoria RAM. In questo modo, se una riga è corrotta, la decodifica fallirà localmente senza bloccare il file stream. Il programma potrà segnalare l'anomalia su `stderr` e passare tranquillamente alla riga successiva.

11.3.4 Strumenti di Tokenizzazione: `sscanf` e `strtok`

Strumento A: La funzione `sscanf`

`sscanf` funziona in modo speculare a `scanf/fscanf`, ma estrae i dati analizzando una stringa in memoria RAM invece che un canale fisico:

```
1 char riga[] = "Programmazione_1 9 27";
2 char materia[50];
3 int cfu, voto;
4
5 if (sscanf(riga, "%49s %d %d", materia, &cfu, &voto) == 3) {
6     printf("Letto: %s (%d CFU) - Voto: %d\\n", materia, cfu, voto);
7 }
```

Strumento B: La funzione `strtok` (In-Place Tokenizer)

Definita in `<string.h>`, `strtok` divide una stringa in sotto-stringhe (token) separate da uno o più caratteri delimitatori (es. la virgola per file CSV).

Attenzione

Attenzione `strtok` è una funzione **distruttiva**: opera *in-place* alterando la stringa originaria. Ogni volta che individua un delimitatore, lo sovrascrive inserendovi il terminatore di stringa `'\\{}\\{}0'`. Per questa ragione, passargli come argomento una stringa costante non modificabile (ad es. `char *s = "Etna,Corso,Catania"`) provocherà un crash immediato per violazione di accesso in scrittura (*Segmentation*

0. Stato Iniziale in RAM

E	t	n	a	,	C	o	r	s	o	,	C	a	t	a	n	i	a	\0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

Puntatore Statico
(Prossima partenza)

1. Chiamata: `strtok(riga, ",")`

E	t	n	a	\0	C	o	r	s	o	,	C	a	t	a	n	i	a	\0
---	---	---	---	----	---	---	---	---	---	---	---	---	---	---	---	---	---	----

↑ tok
(ritorna "Etna")

Puntatore Statico

2. Chiamata: `strtok(NULL, ",")`

E	t	n	a	\0	C	o	r	s	o	\0	C	a	t	a	n	i	a	\0
---	---	---	---	----	---	---	---	---	---	----	---	---	---	---	---	---	---	----

↑ tok
(ritorna "Corso")

Puntatore Statico

3. Chiamata: `strtok(NULL, ",")`

E	t	n	a	\0	C	o	r	s	o	\0	C	a	t	a	n	i	a	\0
---	---	---	---	----	---	---	---	---	---	----	---	---	---	---	---	---	---	----

↑ tok
(ritorna "Catania")

NULL

Figura 11.3: Funzionamento interno di `strtok` in RAM

Fault).

Per mantenere traccia della posizione durante l'analisi, `strtok` adotta una logica specifica:

- Alla **prima chiamata**, si passa la stringa da dividere come primo argomento: `strtok(riga, ",")`.
- Alle **chiamate successive**, si passa `NULL` come primo parametro per indicare alla funzione di riprendere l'analisi dallo stato memorizzato internamente: `strtok(NULL, ",")`.

La Trappola del Terminatore

Cosa stampa l'istruzione `printf` alla fine di questo codice?

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main() {
5     char str[] = "Ciao,sono,io";
6     char *a = strtok(str, ",");
7     while (a != NULL) {
8         printf("%s\n", a);

```

```

9      a = strtok(NULL, ",");
10     }
11     printf("Stringa finale: %s\n", str);
12     return 0;
13 }
```

L'istruzione finale stamperà a schermo solo: **Ciao**. Poiché `strtok` ha modificato la prima virgola in `\{\}\{\}\0`, qualunque funzione standard (come la `printf` con `\{\}%s`) arresterà la lettura all'indirizzo iniziale non appena incontra la prima sentinella `\{\}\{\}\0`, ignorando le parole successive che pure risiedono ancora intatte in memoria RAM.

11.4 Quiz di Auto-valutazione

Quiz 1

Domanda

1. Dove vengono collocate fisicamente in memoria le stringhe passate come argomenti tramite `argv`?

- A) Nel segmento RoData (sola lettura), trattandosi di costanti letterali.
- B) All'inizio assoluto dello Stack, subito prima dello Stack Frame del `main()`.
- C) Nello Heap, allocate dinamicamente dal sistema operativo.
- D) Nel segmento BSS, in quanto variabili globali non inizializzate.

Spiegazione: Le stringhe dei parametri di riga di comando vengono caricate nello Stack dal loader dell'OS prima dell'inizio dell'esecuzione del programma. Questo le rende mutabili (possono essere modificate scrivendo in `argv[1]`).

Risposta corretta: B

Soluzione

Quiz 2

Domanda

2. Cosa accade se si reindirizza l'output di un programma tramite comando `myapp > out.txt` ed il programma riscontra un errore che stampa su `stderr`?

- A) L'errore viene scritto nel file `out.txt` insieme ai dati ordinari.
- B) L'errore viene stampato comunque sullo schermo del terminale.
- C) Il programma viene interrotto istantaneamente dall'OS.
- D) L'errore viene scartato silenziosamente.

Spiegazione: Il reindirizzamento con `>` (o `>>`) devia solo lo standard output (`stdout`), lasciando invariata la destinazione di `stderr` che continua a essere diretta allo schermo del terminale (salvo reindirizzone esplicita con `>>`).

Risposta corretta: B

Soluzione

Quiz 3

Domanda

3. Perché il tipo di ritorno della funzione standard `getchar()` è `int` e non `char`?

- A) Perché deve consentire l'acquisizione di numeri interi completi in formato testuale.
- B) Perché deve poter restituire un valore speciale (EOF, pari a `-1`) distinto da tutti i caratteri validi, evitando che su macchine con `char` con segno il byte `0xFF` venga scambiato per fine file, o che con `char` senza segno EOF si trasformi in 255.
- C) Per uniformità con lo standard POSIX che richiede solo tipi a 32 o 64 bit.
- D) Per evitare overflow aritmetici nello stack frame.

Spiegazione: Se `getchar()` restituisse `char`, lo standard C non specifica se sia con o senza segno. Se fosse signed, il carattere valido 255 (`0xFF`) verrebbe promosso a `-1`, confondendosi con EOF. Se fosse unsigned, EOF (`-1`) verrebbe convertito in 255, impedendo il rilevamento di fine file e causando cicli infiniti.

Risposta corretta: B

Soluzione

Quiz 4

Domanda

4. Cosa succede se si passa una costante di stringa definita con le virgolette (ad es. `char *s = "dati,csv";`) come primo parametro della funzione `strtok`?

- A) La stringa viene duplicata nello Heap e tokenizzata in sicurezza.
- B) La funzione sostituisce la prima virgola con un terminatore senza effetti collaterali.
- C) Il programma subisce un crash istantaneo (*Segmentation Fault*) per tentativo di scrittura su memoria protetta.
- D) Viene restituito un puntatore nullo (`NULL`).

Quiz 7

Domanda

7. Da dove legge i dati la funzione standard `sscanf`?

- A) Dalla tastiera, bloccando il terminale fino alla pressione di Invio.
- B) Da uno stream di file precedentemente aperto con `fopen`.
- C) Dallo standard error.
- D) Da una stringa di caratteri (buffer) già presente in memoria RAM.

Soluzione
Risposta corretta: D
Spiegazione: La prima "s" in `sscanf` sta per "string". Essa analizza il contenuto di una stringa in memoria RAM formattandolo secondo gli specificatori di formato forniti, esattamente come `scanf` fa con lo standard input.

Quiz 8

Domanda

8. Se si apre un file in scrittura con `fopen("dati.txt", "w")` e il file esiste già sul disco, qual è il comportamento del sistema?

- A) Il file viene troncato, ovvero svuotato completamente diventando di dimensione zero.
- B) La funzione fallisce restituendo `NULL`.
- C) I nuovi dati vengono aggiunti in coda a quelli già esistenti.
- D) Viene sollevato un errore di runtime non intercettabile.

Soluzione
Risposta corretta: A
Spiegazione: La modalità "w" (write) cancella l'eventuale contenuto preesistente del file. Per scrivere in coda preservando i dati presenti occorre utilizzare la modalità "a" (append).

Quiz 9

Domanda

9. Perché è strettamente necessario invocare `fclose(fp)` al termine delle operazioni di scrittura su file?

- A) Per rilasciare i puntatori ed evitare integer overflow nello stack.
- B) Perché la scrittura avviene inizialmente in un buffer in memoria RAM gestito dall'OS; `fclose` forza il riversamento (*flush*) fisico dei dati sul disco rigido.

- C) Per consentire a `fscanf` di riposizionare il cursore all'inizio del file.
- D) Per forzare la compilazione del file stream.

Soluzione
Risposta corretta: B
Spiegazione: Per ragioni di prestazioni, le scritture su file sono "bufferizzate". I dati non vengono inviati al disco per ogni singolo carattere. La chiusura del file chiude il descrittore ed esegue il flush del buffer, garantendo l'integrità fisica dei dati memorizzati.

Quiz 10

Domanda

10. Nel dividere una stringa con `strtok("a,b,c", ",")`, in cosa vengono convertite le virgole all'interno dell'array originario?

- A) In caratteri di spazio.
- B) Nel carattere terminatore di stringa `'\0'`.
- C) Rimangono inalterate, la funzione lavora su una copia nascosta.
- D) Vengono eliminate compattando la stringa in memoria.

Soluzione
Risposta corretta: B
Spiegazione: `strtok` opera *in-place* e sostituisce fisicamente ciascun delimitatore individuato con il carattere sentinella `'\0'`. Questo spezza l'array in più sotto-stringhe disgiunte ma allocate nella stessa area di memoria RAM.

11.5 Esercizi Pratici

Esercizio 1: Calcolatrice da Riga di Comando

Domanda

(`cli\{}_calc.c`)

Si scriva un programma in C che accetti esattamente tre argomenti da riga di comando: due numeri interi e un operatore matematico (+, -, x). Il programma deve stampare a schermo il risultato dell'operazione. Se il numero di argomenti è errato o l'operatore non è supportato, stampare un opportuno messaggio d'errore su `stderr` e terminare con exit code 1.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6      if (argc != 4) {
7          fprintf(stderr, "Usage: %s [n1] [op] [n2]\n", argv[0]);
8          return 1;
9      }
10     int n1 = atoi(argv[1]), n2 = atoi(argv[3]);
11     if (strcmp(argv[2], "+") == 0) printf("%d\n", n1 + n2);
12     else if (strcmp(argv[2], "-") == 0) printf("%d\n", n1 - n2);
13     else if (strcmp(argv[2], "x") == 0) printf("%d\n", n1 * n2);
14     else if (strcmp(argv[2], "Op errato\n") != 0) {
15         return 0;
16     }

```

Soluzione**Esercizio 2: Funzione di Lettura Robusta Interi****Domanda**

(robust_{}_input.c)

Si scriva una funzione C `int read\_{}_int(const char *prompt, int *val)` che implementi l'acquisizione robusta di un numero intero tramite `fgets` e `sscanf`.

```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4  int read_int(const char *prompt, int *val) {
5      char buf[128];
6      while (true) {
7          printf("%s", prompt);
8          if (!fgets(buf, stdin)) return 0;
9          if (sscanf(buf, "%d", val) == 1) return 1;
10         fprintf(stderr, "Input non valido\n");
11     }
12 }

```

Soluzione**Esercizio 3: Parser In-Memory di Configurazione****Domanda**

(config_{}_parser.c)

Si realizzi un programma che legga coppie chiave=valore saltando commenti (`\{\}`).

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main()
5 {
6     char line[256];
7     while (fgets(line, sizeof(line), stdin)) {
8         if (line[0] == '#' || line[0] == '\n')
9             continue;
10        char *k = strtok(line, "=");
11        char *v = strtok(NULL, "\n");
12        if (k && v) printf("key: %s, Val: %s\n", k, v);
13    }
14    return 0;
15 }

```

Soluzione**Esercizio 4: Filtro CSV****Domanda**

(csv\{}_filter.c)

Si scriva un filtro CSV che stampa solo i cognomi con voto $\geq$ soglia CLI.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, char *argv[]) {
6     long soglia = atol(argv[1]);
7     char line[256];
8     FILE *fp = fopen("studenti.csv", "r");
9     while (fgets(line, sizeof(line), fp)) {
10        char *nome = strtok(line, ",");
11        char *voto_s = strtok(NULL, "\n");
12        if (atol(voto_s) >= soglia) printf("%s\n", nome);
13    }
14    fclose(fp);
15    return 0;
16 }

```

Soluzione**Esercizio 5: Inversione degli argomenti CLI****Domanda**

(cli\{}_reverse.c)

Stampa gli argomenti CLI al contrario, in maiuscolo.

```

1  #include <stdio.h>
2  #include <ctype.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6      for(int i = argc - 1; i > 0; i--) {
7          for(int j = 0; argv[i][j]; j++) putchar(toupper(argv[i][j]))
8          putchar(' ');
9      }
10     return 0;
11 }

```

Soluzione**Esercizio 6: Word & Character Counter****Domanda**

(my\{}_wc.c)

Conta caratteri, parole e righe da stdin.

```

1  #include <stdio.h>
2  #include <ctype.h>
3
4  int main() {
5      int c, chars = 0, words = 0, lines = 0, in_w = 0;
6      while((c = getchar()) != EOF) {
7          chars++;
8          if(c == '\n') lines++;
9          if(isspace(c)) in_w = 0;
10         else if(!in_w) { in_w = 1; words++; }
11     }
12     printf("Righe: %d, Parole: %d, Caratteri: %d\n", lines, words,
13         chars);
14     return 0;
15 }

```

Soluzione**Esercizio 7: Copia File con Error Logging****Domanda**

(file\{}_copy.c)

Si scriva un programma C che accetti due nomi di file come parametri (sorgente e destinazione). Copia il contenuto del primo nel secondo. Gestire tutti i casi di errore (argomenti mancanti, file non apribili) stampando messaggi significativi su stderr e terminando con exit code 1.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) {
5      if (argc != 3) {
6          fprintf(stderr, "Usa: %s [sorgente] [destinazione]\n", argv
          → [0]);
7          return 1;
8      }
9
10     FILE *in = fopen(argv[1], "r");
11     if (in == NULL) {
12         fprintf(stderr, "Errore: impossibile aprire il file
          → sorgente '%s'\n", argv[1]);
13         return 1;
14     }
15     FILE *out = fopen(argv[2], "w");
16     if (out == NULL) {
17         fprintf(stderr, "Errore: impossibile creare il file di
          → destinazione '%s'\n", argv[2]);
18         fclose(in);
19         return 1;
20     }
21     fclose(in);
22
23     int c;
24     while ((c = fgetc(in)) != EOF) {
25         fputc(c, out);
26     }
27     fclose(in);
28     fclose(out);
29     return 0;
30 }
31

```

Soluzione**Esercizio 8: Validatore di Indirizzi IP****Domanda**

(ip\{}_parser.c)

Si scriva un programma C che prenda un indirizzo IP in formato A.B.C.D come parametro e lo stampi in formato esadecimale, validando che gli ottetti siano tra 0 e 255.


```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) {
5      if (argc != 2) {
6          fprintf(stderr, "Uso: %s [indirizzo_IP]\n", argv[0]);
7          return 1;
8      }
9
10     int o[4];
11     char residuo;
12     if (sscanf(argv[1], "%d.%d.%d.%d%c", &o[0], &o[1], &o[2], &o[3], &residuo) != 4) {
13         fprintf(stderr, "Errore: formato IP non valido. Richiesto A
14         ↪ .B.C.D\n");
15         return 1;
16     }
17     for (int i = 0; i < 4; i++) {
18         if (o[i] < 0 || o[i] > 255) {
19             fprintf(stderr, "Errore: ottenuto %d fuori dal
20             ↪ range [0..255]\n", i, o[i]);
21             return 1;
22         }
23     }
24     printf("Hex: %02X.%02X.%02X.%02X\n", o[0], o[1], o[2], o[3]);
25     return 0;
26 }

```

Soluzione**Esercizio 9: Tokenizzazione di Frase****Domanda**

(word\{}_tokenizer.c)

Estrae parole da stringa inserita tramite standard input usando come delimitatori spazi e punteggiatura.

```

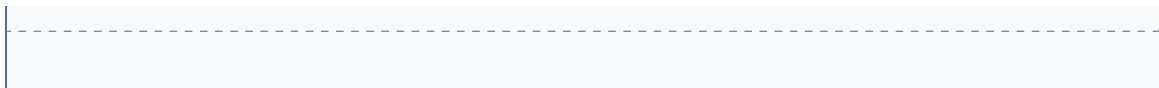
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void) {
5      char s[256];
6      printf("Inserisci una frase: ");
7      if (!fgets(s, sizeof(s), stdin))
8          return 0;
9
10     }
11     char *t = strtok(s, " ,.?! \\n\\t");
12     int i = 1;
13     while (t) {
14         printf("Parola %d: %s\\n", i++, t);
15         t = strtok(NULL, " ,.?! \\n\\t");
16     }
17     return 0;
18 }

```

Soluzione**Esercizio 10: Unione Alternata di File****Domanda**

(file\{}_merge.c)

Prende come parametri due file sorgente ed un file destinazione. Copia
alternativamente una riga del primo e del secondo file nel file destinazione.



```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) {
5      if (argc != 4) {
6          printf(stderr, "Uso: %s [file1] [file2] [destinazione]\n",
7              argv[0]);
8          return 1;
9      }
10     FILE *f1 = fopen(argv[1], "r");
11     if (f1 == NULL) {
12         printf(stderr, "Errore: impossibile aprire '%s'\n", argv
13             [1]);
14         return 1;
15     }
16     FILE *f2 = fopen(argv[2], "r");
17     if (f2 == NULL) {
18         printf(stderr, "Errore: impossibile aprire '%s'\n", argv
19             [2]);
20         fclose(f1);
21         return 1;
22     }
23     FILE *o = fopen(argv[3], "w");
24     if (o == NULL) {
25         printf(stderr, "Errore: impossibile creare '%s'\n", argv
26             [3]);
27         fclose(f1);
28         fclose(f2);
29         return 1;
30     }
31     char l1[256], l2[256];
32     char *r1, *r2;
33     do {
34         r1 = fgets(l1, sizeof(l1), f1);
35         if (r1) {
36             fputs(l1, o);
37         }
38         r2 = fgets(l2, sizeof(l2), f2);
39         if (r2) {
40             fputs(l2, o);
41         }
42     } while (r1 || r2);
43     fclose(f1);
44     fclose(f2);
45     fclose(o);
46     return 0;
47 }

```

LEZIONE 12

Basso Livello, File Binari e l'Abisso della Ricorsione



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 12

Basso Livello, File Binari e l'Abisso della Ricorsione

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

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

12.1 I File Binari e l'Ispezione a Basso Livello

12.1.1 Testo contro Binario: L'Illusione ASCII

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

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

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

12.1.2 La Scrittura Binaria con `fwrite`

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

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

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

Codice C: Confronto tra scrittura testuale e binaria

```

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

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

12.1.3 Hexdump e la Rappresentazione Little Endian

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

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

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

L'output esadecimale mostrerà la seguente configurazione:

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

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

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

12.2 Accesso Casuale e Gestione dei Record Binari

12.2.1 Salvataggio di Array in Blocco

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

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

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

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

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

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

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

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

Il parametro `origin` può assumere tre costanti standard:

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

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

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

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

Codice C: Lettura diretta di un elemento specifico

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

12.2.3 Record a Dimensione Fissa e Stringhe Variabili

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

Tabella

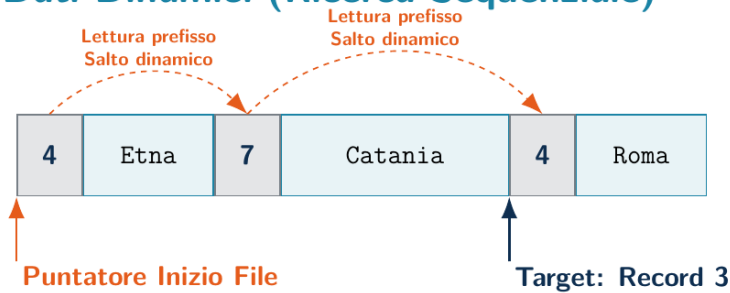
Record			Record			Record		
1	4	"Etna"	2	7	"Catania"	3	4	"Roma"

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

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

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

Dati Dinamici (Ricerca Sequenziale)



Dati a Blocco Fisso (Accesso Diretto con fseek)

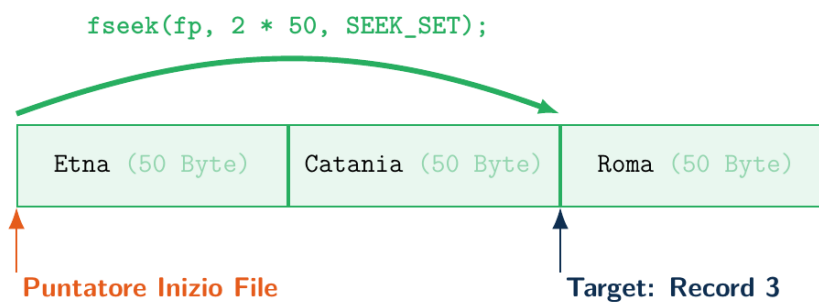


Figura 12.1: Navigazione a Blocchi vs Navigazione Sequenziale

12.3 L'Abisso della Ricorsione e i Limiti dello Stack

12.3.1 I Principi della Ricorsione

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

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

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

12.3.2 Esempi Algoritmici Classici

Il Calcolo del Fattoriale ($N!$)

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

```

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

```

Stampa Inversa di una Stringa

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

```

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

```

La Sequenza di Fibonacci

La definizione ricorsiva della sequenza di Fibonacci prevede:

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

```

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

```

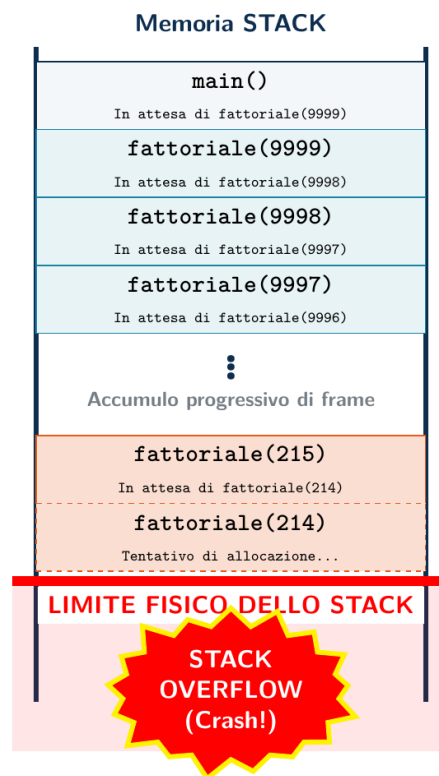


Figura 12.2: Lo Stack Overflow Ricorsivo

12.3.3 Lo Stack Frame e la Catastrofe dello Stack Overflow

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

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

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

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

12.3.4 Inefficienza e la Ricorsione di Coda (Tail Recursion)

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

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

```

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

```

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

La Ricorsione di Coda (Tail Recursion)

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

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

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

```

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

```

Attenzione

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

12.4 Quiz di Auto-valutazione

Quiz 1

Domanda

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

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

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

Quiz 2

Domanda

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

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

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

Quiz 3

Domanda

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

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

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

Quiz 4

Domanda

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

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

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

Quiz 5

Domanda

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

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

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

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

Quiz 6

Domanda

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

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

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

Quiz 7

Domanda

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

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

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

Quiz 8

Domanda

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

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

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

Quiz 9

Domanda

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

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

Soluzione**Risposta corretta: C**

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

Quiz 10**Domanda**

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

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

Soluzione**Risposta corretta: B**

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

12.5 Esercizi Pratici

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

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

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

Soluzione

```

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

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

Esercizio 2: Calcolo della Dimensione di un File

Domanda

(file\{}_size.c)

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

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

```

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

Soluzione

Esercizio 3: Lettore di Record Binario Fissato

Domanda

(record\{}_reader.c)

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

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

--

```

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

```

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

Esercizio 4: Massimo Elemento Ricorsivo

Domanda

(recursive\{}_max.c)

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

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

```

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

Soluzione

Esercizio 5: Palindromia Ricorsiva

Domanda

(recursive\{}_palindrome.c)

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

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

```

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

```

Soluzione

Esercizio 6: Copia Inversa di File Binari

Domanda

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

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

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

```

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

```

Soluzione

Esercizio 7: Fibonacci Tail-Recursive vs Esponenziale

Domanda

(fibonacci\{}_compare.c)

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

--

Soluzione

```

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

```

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

Esercizio 8: Ricerca Binaria Ricorsiva

Domanda

(recursive_binary_search.c)

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

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

```

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

```

Soluzione

Esercizio 9: Hexadecimal File Dumper

Domanda

(hex\{}_dumper.c)

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

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

```

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

Soluzione

Esercizio 10: Moltiplicazione Russa Ricorsiva

Domanda

(russian\{}_multiplication.c)

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

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

```

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

```

Soluzione

Modellare la Realtà (Struct, Padding e Persistenza a Blocchi)



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 13

Modellare la Realtà (Struct, Padding e Persistenza a Blocchi)

Nelle lezioni precedenti abbiamo visto come gestire insiemi di dati omogenei tramite gli array e come persistere le informazioni su file di testo e binari. Tuttavia, la realtà che ci circonda è complessa ed eterogenea: un'entità del mondo reale come uno "Studente" non è definibile tramite un singolo numero o una singola stringa, ma possiede un insieme di caratteristiche di tipo diverso (un nome, un numero di matricola intero, una media pesata float).

In questa lezione supereremo la fragilità degli array paralleli introducendo il concetto di **Struttura (struct)**, che permette di aggregare tipi di dato eterogenei in un unico record logico. Esploreremo la fisica della memoria analizzando come vengono passate le struct alle funzioni (per valore vs riferimento con l'operatore freccia ->) e svelando le bolle di memoria invisibili create dal compilatore per l'efficienza dei bus hardware (**Padding e Allineamento**). Infine, vedremo come le struct rivoluzionano la persistenza binaria consentendo la scrittura e la lettura massiva a blocchi e l'accesso diretto istantaneo tramite la navigazione con `fseek`.

13.1 La Fine degli Array Paralleli e l'Incapsulamento

13.1.1 Il Problema: I limiti dei tipi predefiniti

Fino ad ora, per modellare una tabella di elementi eterogenei, siamo stati costretti ad adottare la tecnica degli **array paralleli**. Se ad esempio desideriamo gestire un elenco di esami universitari composto da materia, CFU e voto, dichiariamo tre array distinti che condividono implicitamente lo stesso indice logico:

```
1 char materie[100][50]; // Nomi degli esami
2 int cfu[100];           // Crediti di ciascun esame
3 int voti[100];          // Voto conseguito
```

Sebbene questa soluzione funzioni, essa presenta gravissimi problemi di manutenibilità e sicurezza dei dati. Consideriamo il caso in cui si voglia ordinare gli esami in base al voto usando l'algoritmo Bubble Sort. Per non mescolare i dati, siamo costretti a effettuare lo scambio (swap) per ciascuno dei tre array paralleli coordinando manualmente gli indici:

```
1 // Scambio del voto
2 int temp_voto = voti[i];
```

```

3  voti[i] = voti[j];
4  voti[j] = temp_voto;
5
6  // Scambio dei CFU
7  int temp_cfu = cfu[i];
8  cfu[i] = cfu[j];
9  cfu[j] = temp_cfu;
10
11 // Scambio della stringa materia
12 char temp_mat[50];
13 strcpy(temp_mat, materie[i]);
14 strcpy(materie[i], materie[j]);
15 strcpy(materie[j], temp_mat);

```

Se il programmatore dimentica di scambiare anche solo una delle variabili (ad esempio omettendo lo swap del nome della materia), l'associazione logica viene irrimediabilmente distrutta, attribuendo ad esempio i CFU di Matematica all'esame di Storia. Il C mette a disposizione tipi primitivi (int, float, char), ma non possiede un tipo nativo denominato Esame. Diventa quindi necessario creare un nostro tipo di dato personalizzato.

13.1.2 La Creazione di un Nuovo Tipo: La struct

In C, la parola chiave `struct` (Struttura) permette di raggruppare variabili di tipo diverso all'interno di un unico contenitore logico e fisico. Ciascuna variabile interna alla struttura viene chiamata **campo** o **membro**.

Definizione di una Struct

La definizione descrive il modello ("progetto") della struttura senza allocare spazio in memoria RAM:

```

1 struct materia_s {
2     char nome[50];
3     char codice[10];
4     int cfu;
5     int voto;
6 }; // ATTENZIONE: Il punto e virgola finale ? obbligatorio!

```

Per istanziare una variabile di questo tipo e accedere ai suoi membri, utilizziamo l'**operatore punto (.)**:

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main(void) {
5     // Dobbiamo ripetere la parola chiave 'struct'
6     struct materia_s analisi;
7
8     // Accesso e scrittura nei campi (i campi array si riempiono con
9     ↪ strcpy)
10    strcpy(analisi.nome, "Analisi");
11    strcpy(analisi.codice, "AN29345");

```

```

11     analisi.cfu = 9;
12     analisi.voto = 28;
13
14     printf("Voto in analisi: %d\n", analisi.voto);
15     return 0;
16 }

```

Definizione Elegante con typedef

Dover anteporre costantemente `struct` davanti al nome del tipo risulta ridondante. Possiamo utilizzare la keyword `typedef` per creare un alias (sinonimo) semplificato:

```

1  typedef struct {
2      char nome[50];
3      char codice[10];
4      int cfu;
5      int voto;
6  } Esame; // "Esame" diventa a tutti gli effetti un nuovo tipo di dato!

```

È anche possibile scorporare la definizione della struct e il typedef:

```

1  struct esame {
2      char nome[50];
3      char codice[10];
4      int cfu;
5      int voto;
6  };
7
8  typedef struct esame Esame; // "Esame" diventa a tutti gli effetti un
    ↪ nuovo tipo di dato!

```

Nota

Nota Per convenzione stilistica diffusa, i tipi personalizzati definiti dal programmatore si scrivono adottando la nomenclatura **PascalCase** (es. `Esame` o `StudenteIngegneria`), distinguendoli a colpo d'occhio dai tipi primitivi nativi (`int`, `float`, `char`) scritti interamente in minuscolo.

13.1.3 Inizializzazione e Assegnazione Magica

Inizializzazione a Graffe

Una struct può essere inizializzata sinteticamente all'atto della dichiarazione usando una lista di valori racchiusi tra parentesi graffe `{...}`, rispettando rigorosamente l'ordine di dichiarazione dei campi:

```

1  // Corretto: {nome, codice, cfu, voto}
2  Esame prog = {"Programmazione 1", "INF101", 12, 30};
3

```

```
4 // Errato o disallineato:  
5 Esame errore = {"Fisica", "FIS01", 18, 6}; // Assegna 18 a cfu e 6 a voto  
    ↪ !
```

Copia per Assegnazione

A differenza degli array tradizionali (che non possono essere duplicati tramite l'operatore =), le struct godono del privilegio della **copia diretta per assegnamento**. Il compilatore effettua autonomamente un dump perfetto bit-a-bit dell'intera area di memoria occupata dalla struct:

```
1 Esame backup = prog; // Copia immediata di tutti i campi in RAM
```

13.2 Puntatori a Struct e Fisica della Memoria

13.2.1 Passaggio per Valore vs Riferimento: L'Operatore Freccia (->)

Sommando lo spazio occupato dai campi di Esame ricaviamo la sua dimensione: 50 (nome) + 10 (codice) + 4 (cfu) + 4 (voto) = 68 byte.

Passaggio per Valore (Inefficiente)

Se passiamo una struct a una funzione per valore, l'intera struttura da 68 byte viene interamente duplicata sullo Stack all'interno del record di attivazione della funzione chiamata:

```
1 void stampa_copia(Esame e) {  
2     printf("%s: %d\n", e.nome, e.voto);  
3 }
```

Qualora la struct contenesse array di dimensioni maggiori, questa operazione comporterebbe un pesante spreco di memoria Stack e cicli di clock per effettuare la fotocopia dei byte.

Passaggio per Riferimento (Efficiente)

Per ottimizzare le prestazioni, passiamo unicamente l'indirizzo di memoria della struct (8 byte su sistemi a 64-bit). Per accedere ai campi della struct puntata abbiamo due opzioni sintattiche:

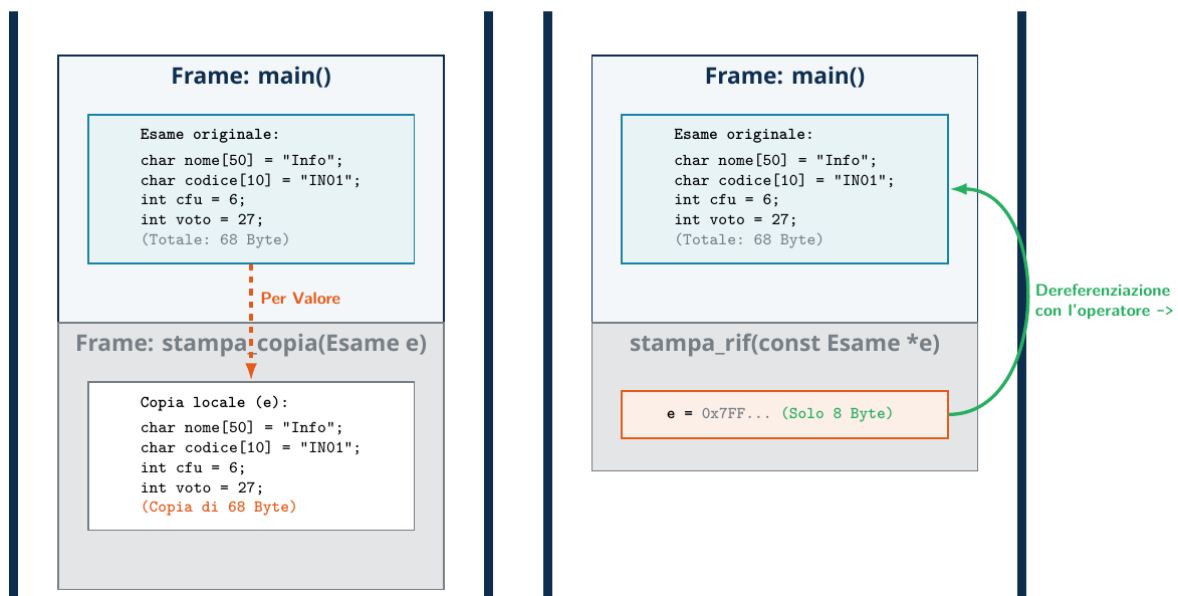
1. **Sintassi Esplicita**: Dereferenziare il puntatore e usare l'operatore punto. Richiede le parentesi a causa della precedenza degli operatori: `(*e).nome`.
2. **Sintassi Semplificata (Operatore Freccia ->)**: Zucchero sintattico che sostituisce la combinazione di dereferenziazione e punto: `e->nome`.

```

1  #include <stdio.h>
2
3  void stampa_rif(const Esame *e) {
4      // Accesso tramite operatore freccia ->
5      printf("%s (%s) - CFU: %d, Voto: %d\n", e->nome, e->codice, e->cfu, e
        ↪ ->voto);
6  }
7
8  int main(void) {
9      Esame info = {"Informatica", "INF01", 6, 27};
10     stampa_rif(&info); // Passiamo l'indirizzo
11     return 0;
12 }

```

Scenario A: Passaggio per Valore Scenario B: Passaggio per Riferimento



Suggerimento

Regola Pratica Se possiedi direttamente la variabile struct (l'oggetto in sé) utilizza l'operatore punto `..`. Se possiedi un puntatore alla struct, utilizza l'operatore freccia `->`.

13.2.2 Sotto il Cofano: Padding e Allineamento (Data Alignment)

Consideriamo l'esperimento di verificare lo spazio occupato in RAM da due struct apparentemente identiche nei campi, ma disposte in ordine differente:

```

1  typedef struct {
2      char c;      // 1 byte
3      double d;    // 8 byte
4      int i;       // 4 byte
5  } ProvaA;
6

```

```
7 typedef struct {  
8     double d;    // 8 byte  
9     int i;       // 4 byte  
10    char c;      // 1 byte  
11 } ProvaB;
```

A livello algebrico, lo spazio occupato dai campi dovrebbe essere pari a $1 + 8 + 4 = 13$ byte. Tuttavia, compilando ed eseguendo `sizeof` su una macchina a 64-bit ricaviamo risultati sorprendenti:

- `sizeof(ProvaA)` restituisce **24 byte**.
- `sizeof(ProvaB)` restituisce **16 byte**.

Il Padding

Questo fenomeno è causato dal **Padding** (imbottitura) introdotto dal compilatore. I moderni processori leggono la memoria RAM a blocchi allineati (solitamente di 8 byte su sistemi a 64-bit). Se una variabile multi-byte (come un `double` da 8 byte) venisse memorizzata a cavallo tra due blocchi fisici di memoria, la CPU sarebbe costretta a eseguire due letture RAM distinte per ricomporre il dato, rallentando l'esecuzione.

Per ottimizzare gli accessi hardware, il compilatore applica le seguenti regole di allineamento:

1. Ogni tipo di dato nativo deve iniziare a un indirizzo di memoria che sia multiplo della sua stessa dimensione (es. un `double` allineato a indirizzi multipli di 8, un `int` a multipli di 4).
2. Il compilatore inietta byte vuoti (padding) invisibili per ripristinare l'allineamento ottimale.

Analizziamo il posizionamento dei byte per le due strutture:

- **ProvaA (char c, double d, int i):**
 - Il carattere `c` occupa la posizione 0.
 - Il campo successivo `d` (double) richiede allineamento a 8 byte. Il compilatore inserisce **7 byte di padding** vuoti.
 - Il `double` viene allocato dall'offset 8 all'offset 15.
 - L'intero `i` viene allocato dall'offset 16 al 19 (allineamento a 4 rispettato).
 - La dimensione totale della struct deve essere multipla del campo più grande (`double` = 8). Vengono iniettati altri **4 byte di padding** finali, portando il totale a **24 byte** (offset 0-23).
- **ProvaB (double d, int i, char c):**
 - Il `double` viene allocato al blocco iniziale dall'offset 0 al 7.

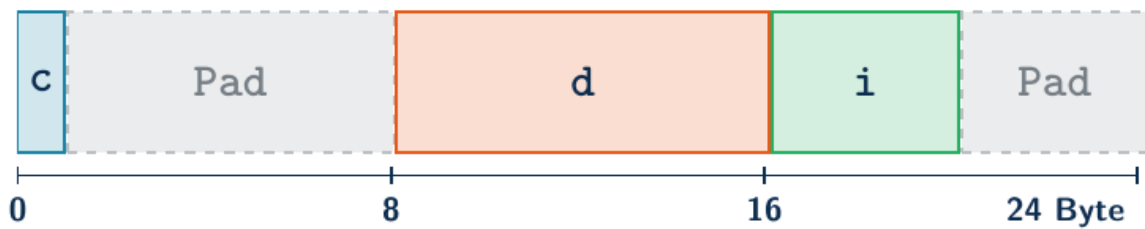
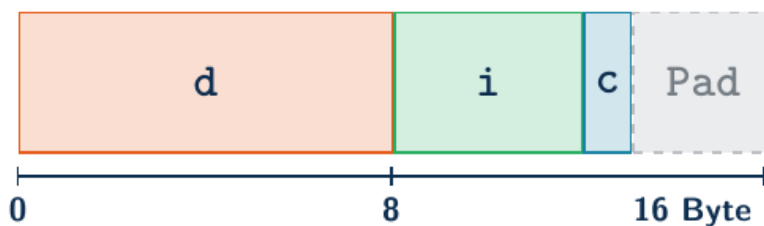
struct ProvaA (Dimensione Totale = 24 Byte)**struct ProvaB (Dimensione Totale = 16 Byte)**

Figura 13.1: Padding e Allineamento

- L'intero **i** occupa la posizione dall'offset 8 all'11 (allineamento a 4 rispettato).
- Il carattere **c** occupa l'offset 12.
- Vengono iniettati **3 byte di padding** finali per allineare l'intera struttura al multiplo di 8, portando il totale a **16 byte** (offset 0-15).

Attenzione

Attenzione Nei normali esercizi di laboratorio non è richiesto riordinare ossessivamente le struct per risparmiare memoria. È tuttavia fondamentale, in contesti a basso livello (sistemi operativi, sistemi embedded e programmazione di rete), conoscere l'esistenza del padding per prevenire anomalie e sprechi di RAM.

13.2.3 Enumerazioni (enum): Un tocco di leggibilità

Spesso all'interno delle struct è necessario salvare lo stato di un oggetto (es. un esame può essere "In Corso", "Superato" o "Bocciato"). Utilizzare numeri interi generici (0, 1, 2) rende il codice cryptico ed esposto a errori.

Il C permette di creare delle **Enumerazioni (enum)**, che associano identificatori mnemonici a costanti intere incrementali (di default partendo da 0):

```

1 #include <stdio.h>
2
3 // Definizione dell'enumerazione
4 typedef enum {
5     IN_CORSO,    // 0

```

```
6     SUPERATO,    // 1
7     BOCCIATO    // 2
8 } StatoEsame;
9
10 // Utilizzo all'interno di una Struct
11 typedef struct {
12     char materia[50];
13     StatoEsame stato;
14 } EsameCarriera;
15
16 int main(void) {
17     EsameCarriera prog = {"Programmazione 1", IN_CORSO};
18
19     if (prog.stato == IN_CORSO) {
20         printf("Stai ancora studiando %s. Coraggio!\n", prog.materia);
21     }
22
23     prog.stato = SUPERATO; // Aggiornamento dello stato
24     return 0;
25 }
```

13.3 Il Trionfo di fread, fwrite ed fseek

13.3.1 Unire la Teoria: Struct e File Binari

Nella Lezione 12 abbiamo notato la difficoltà di gestire file binari composti da stringhe e dati a lunghezza variabile: senza una dimensione nota e costante, risulta impossibile calcolare l'offset fisico per saltare a un record specifico.

La struct rappresenta il mattoncino fondamentale per risolvere questo problema: poiché la dimensione di una struct (ad esempio `sizeof(Esime)`) è fissa e nota a tempo di compilazione, un file binario composto da una successione di struct si comporterà fisicamente come un array sequenziale su disco.

13.3.2 Il Salvataggio Istantaneo dell'intero Database

Grazie alla contiguità fisica della memoria RAM, possiamo salvare un intero array di struct su disco con una singola invocazione di `fwrite`, riversando i blocchi in un unico colpo:

```
1 #include <stdio.h>
2
3 typedef struct {
4     char nome[50];
5     int cfu;
6     int voto;
7 } Esame;
8
9 int main(void) {
10     // Array di struct in RAM
11     Esame libretto[3] = {
12         {"Analisi 1", 9, 28},
13         {"Programmazione 1", 12, 30},
```

```

14     {"Fisica 1", 6, 25}
15 };
16
17 FILE *fp = fopen("database.bin", "wb");
18 if (fp != NULL) {
19     // Scrittura in blocco: (sizeof(Esame) * 3) byte trasferiti sul
    ↪ disco
20     fwrite(libretto, sizeof(Esame), 3, fp);
21     fclose(fp);
22     printf("Database salvato con successo!\n");
23 }
24 return 0;
25 }

```

13.3.3 Accesso Diretto

Grazie alla struttura a blocchi costanti, la testina di lettura può saltare istantaneamente all'inizio del record N -esimo senza dover scansionare i record precedenti. L'offset esatto in byte a partire dall'inizio del file (`SEEK\{\}_SET`) è calcolato con la formula:

$$\text{Offset} = \text{indice} \times \text{sizeof}(\text{Struct})$$

```

1  #include <stdio.h>
2
3  typedef struct {
4      char nome[50];
5      int cfu;
6      int voto;
7  } Esame;
8
9  int main(void) {
10     FILE *fp = fopen("database.bin", "rb");
11     if (fp == NULL) {
12         return 1;
13     }
14
15     Esame cercato;
16     int indice_desiderato = 1; // Vogliamo leggere il secondo esame (
    ↪ indice 1)
17
18     // SALTO MATEMATICO
19     fseek(fp, indice_desiderato * sizeof(Esame), SEEK_SET);
20
21     // Lettura isolata dell'unico record posizionato sotto la testina
22     fread(&cercato, sizeof(Esame), 1, fp);
23     fclose(fp);
24
25     printf("Esame estratto al volo: %s, Voto: %d\n", cercato.nome,
    ↪ cercato.voto);
26     return 0;
27 }

```

File Binario database.bin (Blocchi da 68 Byte)

```
fseek(fp, 1 * sizeof(Esame), SEEK_SET);
```



Figura 13.2: I/O Diretto con Array di Struct su File Binario

Nota: il diagramma riporta blocchi da 68 byte (dimensione della Esame con il campo codice); con la Esame di questa sezione, priva di codice, sizeof(Esame) vale 60 byte su x86-64 (50 + 2 di padding + 4 + 4), quindi il record di indice 1 inizia all'offset 60.

13.4 Quiz di Auto-valutazione

Quiz 1

Domanda

1. Quale delle seguenti affermazioni sulla definizione di una struct in C è corretta?

- A) La definizione alloca immediatamente spazio nella memoria RAM.
- B) È obbligatorio includere il punto e virgola ; dopo la chiusura della parentesi graffa.
- C) Permette di raggruppare unicamente elementi dello stesso tipo.

Spiegazione: La definizione di una struct funge solo da modello ("stampo") per il compilatore e non alloca memoria (che viene allocata solo all'istanza delle variabili). La sintassi del linguaggio C impone tassativamente il punto e virgola ; al termine della definizione della struttura.

Risposta corretta: B

Soluzione

Quiz 2

Domanda

2. Cosa accade quando si esegue l'operazione di assegnamento structA = structB in C?

- A) Viene copiato solo l'indirizzo di memoria del primo elemento (come per gli array).

- B) Il compilatore restituisce un errore, poiché le struct non si possono copiare con =.
- C) Viene effettuata una copia fisica bit-a-bit di tutti i campi in una nuova porzione di RAM.

Risposta corretta: C
Spiegazione: A differenza degli array, le struct in C sono trattate come valori atomici per l'assegnamento. Il compilatore genera codice per copiare l'intera porzione di memoria occupata dalla struct sorgente in quella di destinazione.

Soluzione

Quiz 3

Domanda

3. Data la struct `typedef struct \{\} { int x; char y; \{\}} Punto; Punto p; Punto *ptr = \{\}&p;`, qual è la sintassi corretta per accedere al campo y tramite il puntatore?

- A) `ptr->y`
- B) `*ptr.y`
- C) `ptr.y`

Risposta corretta: A
Spiegazione: L'operatore freccia `->` è lo zucchero sintattico utilizzato per accedere ai membri di una struttura puntata da un puntatore. Scrivere `ptr->y` equivale a scrivere `(*ptr).y`. La scrittura `*ptr.y` senza parentesi è errata poiché l'operatore punto `.` ha una precedenza maggiore rispetto all'operatore di dereferenziazione `*`.

Soluzione

Quiz 4

Domanda

4. Perché `sizeof(struct \{\} { char c; double d; \{\}})` restituisce solitamente 16 anziché 9?

- A) A causa dell'allineamento dei dati (padding) effettuato dal compilatore per l'efficienza della CPU.
- B) Perché il compilatore C riserva byte aggiuntivi per i nomi delle variabili.
- C) Perché il tipo `char` viene convertito automaticamente in `double`.

Soluzione
Risposta corretta: A
Spiegazione: Le CPU a 64-bit accedono in modo efficiente a dati situati ad indirizzi multipli di 8. Per evitare accessi disallineati, il compilatore inserisce 7 byte di padding vuoti subito dopo il char da 1 byte, allineando il double (8 byte) ai blocchi hardware della CPU.

Quiz 5

Domanda

5. Cos'è una enum in C?

- A) Un tipo di dato che alloca dinamicamente spazio nello Stack.
- B) Un insieme di costanti intere con nomi mnemonici per migliorare la leggibilità del codice.
- C) Un file binario in cui memorizzare le struct.

Soluzione
Risposta corretta: B
Spiegazione: Le enumerazioni consentono di mappare nomi mnemonici a valori interi sequenziali (es. 0, 1, 2). Sotto il cofano, le variabili di tipo enum sono gestite dal compilatore come normali numeri interi, ma prevenengono l'uso di "numeri magici" sparsi nel codice.

Quiz 6

Domanda

6. Se definiamo `typedef struct \{\}\ char nome[20]; \{\}\ Persona;`, quanti byte occuperà un puntatore `Persona *p` in un sistema a 64-bit?

- A) 20 byte.
- B) 8 byte.
- C) 4 byte.

Soluzione
Risposta corretta: B
Spiegazione: La dimensione di un puntatore dipende esclusivamente dall'architettura del processore (indirizzamento a 64-bit = 8 byte, 32-bit = 4 byte), indipendentemente dal tipo di dato o dalla dimensione della struttura a cui punta.

Quiz 7

Domanda

7. Qual è la formula matematica per calcolare l'offset in byte per raggiungere il record N -esimo (con indice 0-based) all'interno di un file binario strutturato con struct di tipo Record?

- A) $N * \text{sizeof}(\text{Record})$
- B) $(N - 1) * \text{sizeof}(\text{Record})$
- C) $N * \text{sizeof}(\text{Record}) + 1$

Soluzione
Risposta corretta: A
Spiegazione: Avendo i record una dimensione fissa costante, il record all'indice N si trova esattamente dopo che si sono saltati N record interi di dimensione $\text{sizeof}(\text{Record})$. Per l'indice 0, lo spostamento sarà di 0 byte (l'inizio del file).

Quiz 8

Domanda

8. Per quale motivo gli array paralleli sono considerati una pratica fragile in programmazione C?

- A) Perché il compilatore impedisce la compilazione di più di due array contemporaneamente.
- B) Perché richiedono la gestione sincrona manuale degli indici; se si effettua uno swap su un solo array, i dati degli altri si disallineano corrompendo l'informazione.
- C) Perché consumano il doppio dello spazio in memoria rispetto a una struct equivalente.

Soluzione
Risposta corretta: B
Spiegazione: Gli array paralleli non garantiscono l'integrità del dato a livello di compilatore. Qualsiasi operazione di ordinamento o inserimento costringe a manipolare simultaneamente tutti gli array; un singolo errore disallinea irrimediabilmente i record.

Quiz 9

Domanda

9. Con quale costante in `fseek` si indica che lo spostamento deve avvenire a partire dalla posizione corrente del cursore?

- A) `SEEK\{\}_SET`

- B) SEEK\{}_END
- C) SEEK\{}_CUR

Soluzione
Risposta corretta: C
Spiegazione: In fseek, SEEK\{}_SET indica l'inizio del file, SEEK\{}_END indica la fine del file, e SEEK\{}_CUR indica la posizione corrente in cui si trova la testina di lettura/scrittura.

Quiz 10

Domanda

10. Qual è l'effetto di omettere il punto e virgola ; alla fine della definizione di una struct?

- A) Nessuno, il compilatore lo aggiunge automaticamente.
- B) Un errore di compilazione ("syntax error").
- C) La struct viene allocata nello Heap invece che nello Stack.

Soluzione
Risposta corretta: B
Spiegazione: La sintassi formale del linguaggio C richiede obbligatoriamente il punto e virgola al termine della definizione di una struttura. In caso contrario, il compilatore genera un errore bloccante.

13.5 Esercizi Pratici

Esercizio 1: Definizione e Stampa

Domanda

(stampa\{}_punto.c)

Si definisca una struct Punto3D avente tre coordinate reali (x, y, z). Si scriva una funzione stampa\{}_punto che accetti la struct per valore e ne stampi le coordinate nel formato (x, y, z).

Spiegazione dell'algoritmo: Definiamo il tipo Punto3D contenente tre variabili double. La funzione stampa_punto\{\}_punto riceve una copia locale della struttura e stampa i singoli campi accedendovi con l'operatore punto (.).

```

1  #include <stdio.h>
2
3  // Definizione della struct
4  typedef struct {
5      double x;
6      double y;
7      double z;
8  } Punto3D;
9
10 // Funzione che accetta la struct per valore
11 void stampa_punto(Punto3D p) {
12     printf("%.2f, %.2f, %.2f\n", p.x, p.y, p.z);
13 }
14
15 int main(void) {
16     Punto3D origine = {0.0, 0.0, 0.0};
17     Punto3D p1 = {3.5, -2.0, 7.8};
18     printf("Origine: ");
19     stampa_punto(origine);
20
21     printf("Punto 1: ");
22     stampa_punto(p1);
23
24     return 0;
25 }
26

```

Soluzione

Esercizio 2: Modifica per Riferimento

Domanda

(trasla\{\}_punto.c)

Si scriva una funzione trasla\{\}_punto che accetti un puntatore a Punto3D e tre valori reali dx, dy, dz per traslare il punto sommandoli alle coordinate originali.

Spiegazione dell'algoritmo: Passando il puntatore Punto3D *p, la funzione può accedere e modificare le coordinate reali del punto originale allocato nel frame del main tramite l'operatore freccia ->, evitando di creare copie ridondanti in memoria Stack.

```

1  #include <stdio.h>
2
3  typedef struct {
4      double x;
5      double y;
6      double z;
7  } Punto3D;
8
9  // Modifica tramite puntatore e operatore freccia ->
10 void trasla_punto(Punto3D *p, double dx, double dy, double dz) {
11     p->x += dx;
12     p->y += dy;
13     p->z += dz;
14 }
15
16 void stampa_punto(Punto3D p) {
17     printf("%.2f, %.2f, %.2f\n", p.x, p.y, p.z);
18 }
19
20 int main(void) {
21     Punto3D p = {1.0, 2.0, 3.0};
22
23     printf("Punto originale: ");
24     stampa_punto(p);
25
26     // Passiamo l'indirizzo di memoria di p
27     trasla_punto(&p, 1.5, -0.5, 2.0);
28
29     printf("Punto traslato: ");
30     stampa_punto(p);
31
32     return 0;
33 }

```

Soluzione

Esercizio 3: Calcolo Statistiche su Array di Struct

Domanda

(statistiche\{}_studenti.c)

Si definisca una struct `Studente` contenente il nome (stringa max 50 caratteri) e la media dei voti (valore float). Si scriva un programma che istanzi un array di 3 studenti, legga i dati da tastiera e visualizzi la media complessiva della classe.

Spiegazione dell'algoritmo: Creiamo un array in cui ogni cella contiene una struct studente. Nel ciclo for accediamo ai campi di ciascun elemento combinando l'indice dell'array e l'operatore punto: `classe[i].nome` e `\{\}&classe[i].media` (notare l'operatore indirizzamento `\{\}&` posizionato davanti al membro per leggerlo con `scanf`).

```

1  #include <stdio.h>
2  #include <string.h>
3
4  typedef struct {
5      char nome[50];
6      float media;
7  } Studente;
8
9  int main(void) {
10     Studente classe[3];
11     float somma = 0.0;
12
13     for (int i = 0; i < 3; i++) {
14         printf("Inserisci il nome dello studente %d: ", i + 1);
15         // Lettura stringa con gestione del fine riga
16         scanf("%49s", classe[i].nome);
17
18         printf("Inserisci la media dello studente %d: ", i + 1);
19         scanf("%f", &classe[i].media);
20
21         somma += classe[i].media;
22     }
23
24     float media_generale = somma / 3.0;
25     printf("\nMedia generale della classe: %.2f\n", media_generale);
26
27     return 0;
28 }

```

Soluzione

Esercizio 4: Ricerca per Chiave in Archivio

Domanda

(cerca\{\}_studente.c)

Aggiungere all'esercizio precedente un campo intero matricola. Si realizzi una funzione che, dato un array di studenti e una matricola da cercare, restituisca l'indice dello studente trovato, oppure -1 se non presente.

Spiegazione dell'algoritmo: Scansiamo sequenzialmente l'array di struct. Ad ogni iterazione, confrontiamo il valore della variabile cercata con il campo elenco[i].matricola. Non appena riscontriamo uguaglianza, interrompiamo restituendo la posizione corrente dell'array.

```

1  #include <stdio.h>
2
3  typedef struct {
4      char nome[50];
5      int matricola;
6      float media;
7  } Studente;
8
9  // Funzione di ricerca lineare
10 int cerca_studente(const Studente elenco[], int n, int
    ↪ matricola_cercata) {
11     for (int i = 0; i < n; i++) {
12         if (elenco[i].matricola == matricola_cercata) {
13             return i; // Restituisce l'indice della corrispondenza
14         }
15     }
16     return -1; // Non trovato
17 }
18
19 int main(void) {
20     Studente archivio[3] = {
21         {"Alice", 104523, 28.5},
22         {"Bob", 104589, 24.2},
23         {"Charlie", 104612, 27.8}
24     };
25
26     int matr = 104589;
27     int idx = cerca_studente(archivio, 3, matr);
28
29     if (idx != -1) {
30         printf("Studente trovato: %s, Media: %.2f\n", archivio[idx
    ↪ ].nome, archivio[idx].media);
31     } else {
32         printf("Nessuno studente con matricola %d in archivio.\n",
    ↪ matr);
33     }
34
35     return 0;
36 }

```

Soluzione

Esercizio 5: Scambio e Ordinamento di Struct

Domanda

(ordina\{}_studenti.c)

Si scriva una funzione per scambiare due struct `Studiante` e la si utilizzi per ordinare in senso decrescente (in base alla media) l'array di studenti precedentemente creato.

Spiegazione dell'algoritmo: La funzione `scambia` sfrutta la caratteristica nativa del C di copiare per intero le struct tramite assegnamento (`temp = *a;`). Nel ciclo del Bubble Sort, verifichiamo la condizione sulla media e scambiamo le intere strutture dati con un'unica chiamata, mantenendo l'allineamento dei record.

```

1  #include <stdio.h>
2
3  typedef struct {
4      char nome[50];
5      int matricola;
6      float media;
7  } Studiante;
8
9  // Funzione di swap per intere struct
10 void scambia_studenti(Studiante *a, Studiante *b) {
11     // Sfruttiamo la copia diretta del C
12     Studiante temp = *a;
13     *a = *b;
14     *b = temp;
15 }
16
17 int main(void) {
18     Studiante archivio[3] = {
19         {"Bob", 104589, 24.2},
20         {"Alice", 104523, 28.5},
21         {"Charlie", 104612, 27.8}
22     };
23
24     // Ordinamento Bubble Sort
25     for (int i = 0; i < 3 - 1; i++) {
26         for (int j = 0; j < 3 - i - 1; j++) {
27             if (archivio[j].media < archivio[j+1].media) {
28                 // Passiamo gli indirizzi per lo swap
29                 scambia_studenti(&archivio[j], &archivio[j+1]);
30             }
31         }
32     }
33
34     printf("Archivio ordinato per media decrescente:\n");
35     for (int i = 0; i < 3; i++) {
36         printf("%s (Matr: %d) - Media: %.2f\n", archivio[i].nome,
37             ↪ archivio[i].matricola, archivio[i].media);
38     }
39     return 0;
40 }

```

Soluzione

Esercizio 6: Gestione di Stati tramite**Domanda**

enum(carriera\{}_esami.c)

Si definisca una struct Corso contenente il nome della materia e un campo stato di tipo enum (DA\{}_SOSTENERE, SUPERATO, FALLITO). Si scriva una funzione che stampi a schermo l'elenco dei corsi colorando simbolicamente l'output a seconda dello stato.

Soluzione

```

1 #include <stdio.h>
2
3 typedef enum {
4     DA_SOSTENERE,
5     SUPERATO,
6     FALLITO
7 } StatoCorso;
8
9 typedef struct {
10     char materia[50];
11     StatoCorso stato;
12 } Corso;
13
14 void stampa_libretto(const Corso libretto[], int n) {
15     for (int i = 0; i < n; i++) {
16         printf("Materia: %-20s -> Stato: ", libretto[i].materia);
17         switch (libretto[i].stato) {
18             case DA_SOSTENERE:
19                 printf("[ ] Da Sostenere\n");
20                 break;
21             case SUPERATO:
22                 printf("[X] Superato\n");
23                 break;
24             case FALLITO:
25                 printf("[!] Fallito\n");
26                 break;
27         }
28     }
29 }
30
31 int main(void) {
32     Corso esami[3] = {
33         {"Programmazione 1", SUPERATO},
34         {"Architettura", DA_SOSTENERE},
35         {"Fisica 1", FALLITO}
36     };
37     stampa_libretto(esami, 3);
38     return 0;
39 }
40

```

Spiegazione dell'algoritmo: Il tipo `StatoCorso` mappa i tre possibili valori simbolici. Nella funzione `stampa_libretto` utilizziamo un costrutto `switch` sul campo `enum libretto[i].stato` per determinare in modo leggibile quale descrizione testuale stampare.

Esercizio 7: Persistenza a Blocchi su Disco

Domanda

(salva\{}_archivio.c)

Si scriva un programma C che legga un array statico di 3 struct `Studiante` e lo scriva integralmente in formato binario all'interno del file `studenti.bin` in un solo colpo tramite la funzione `fwrite`.

Spiegazione dell'algoritmo: Apriamo il file con modalità "wb" (write binary). La funzione `fwrite` riceve come primo parametro il puntatore alla base dell'array `elenco`, e copia sul file $3 \times \text{sizeof}(\text{Studiante})$ byte consecutivi, effettuando il dump della RAM su disco.

```

1  #include <stdio.h>
2
3  typedef struct
4  {
5      char nome[50];
6      int matricola;
7      float media;
8  } Studiante;
9
10 int main(void) {
11     Studiante elenco[3] = {
12         {"Alice", 104523, 28.5},
13         {"Bob", 104589, 24.2},
14         {"Charlie", 104612, 27.8}
15     };
16     FILE *fp = fopen("studenti.bin", "wb");
17     if (fp == NULL) {
18         fprintf(stderr, "Errore di apertura del file in scrittura.\n");
19         return 1;
20     }
21     // Salvataggio massivo dell'intero array in RAM sul file stream
22     size_t scritti = fwrite(elenco, sizeof(Studiante), 3, fp);
23     fclose(fp);
24     printf("Salvati correttamente %zu record di dimensione %zu byte\n", scritti, sizeof(Studiante));
25     return 0;
26 }

```

Soluzione

Esercizio 8: Recupero Blocco Dati da File Binario

Domanda

(carica\{}_archivio.c)

Si realizzi il programma speculare al precedente: il programma apre `studenti.bin` in

lettura binaria, carica l'intero blocco di dati all'interno di un array locale in RAM tramite `fread` e ne stampa a schermo le informazioni.

Spiegazione dell'algoritmo: Utilizziamo `fread` fornendo l'indirizzo dell'array `archivio_letti` in RAM la struttura originale delle struct.

```

1  #include <stdio.h>
2
3  typedef struct {
4      char nome[50];
5      int matricola;
6      float media;
7  } Studente;
8
9  int main(void) {
10     Studente archivio_letti[3];
11
12     FILE *fp = fopen("studenti.bin", "rb");
13     if (fp == NULL) {
14         fprintf(stderr, "Errore di apertura del file in lettura.
15         ↪ Eseguire prima il salvataggio.\n");
16     }
17     // Caricamento massivo del record dal file binario
18     size_t letti = fread(archivio_letti, sizeof(Studente), 3, fp);
19     fclose(fp);
20
21     printf("Caricati con successo %zu record.\n", letti);
22     for (int i = 0; i < 3; i++) {
23         printf("Studente %d: %s (Matr: %d) -> Media: %.2f\n",
24             i + 1, archivio_letti[i].nome, archivio_letti[i].
25             ↪ matricola, archivio_letti[i].media);
26     }
27     return 0;
28 }
29

```

Soluzione

Esercizio 9: Accesso Casuale a Record di Struct

Domanda

(`leggi\{}_record.c`)

Si scriva un programma C che consenta all'utente di inserire da tastiera un indice N (0, 1, 2) e, tramite `fseek`, legga direttamente ed esclusivamente il record corrispondente del file `studenti.bin`, senza caricare in memoria gli altri record.

Soluzione

```

1  #include <stdio.h>
2
3  typedef struct {
4      char nome[50];
5      int matricola;
6      float media;
7  } Studente;
8
9  int main(void) {
10     FILE *fp = fopen("studenti.bin", "rb");
11     if (fp == NULL) {
12         fprintf(stderr, "Errore: impossibile leggere 'studenti.bin'
        ↪ '.\n");
13     }
14     return 1;
15 }
16 int indice = 0;
17 printf("Inserisci l'indice del record da caricare (0, 1, o 2):
        ↪ ");
18 if (scanf("%d", &indice) != 1 || indice < 0 || indice > 2) {
19     printf("Indice non valido!\n");
20     fclose(fp);
21     return 1;
22 }
23 // Salto mirato: calcoliamo l'offset matematico
24 long offset = (long)(indice * sizeof(Studente));
25 fseek(fp, offset, SEEK_SET);
26 // Lettura del singolo record isolato
27 Studente cercato;
28 size_t letti = fread(&cercato, sizeof(Studente), 1, fp);
29 fclose(fp);
30 if (letti == 1) {
31     printf("\n[RECORD %d ESTRATTO]\n", indice);
32     printf("Nome: %s\n", cercato.nome);
33     printf("Matricola: %d\n", cercato.matricola);
34     printf("Media: %.2f\n", cercato.media);
35     } else {
36         printf("Errore durante la lettura del record.\n");
37     }
38     return 0;
39 }
40
41
42
43 }

```

Spiegazione dell'algoritmo: Sfruttando la dimensione fissa di `Studente` nota a tempo di compilazione, posizioniamo la testina di lettura all'offset in byte pari a `indice * sizeof(Studente)`. In questo modo la chiamata a `fread` estrae esclusivamente i byte associati allo studente richiesto.

Esercizio 10: Aggiornamento Diretto su File Binario**Domanda**

(aggiorna\{}_record.c)

Si realizzi una semplice utility C che consenta di modificare in loco la media di un determinato studente salvato su `studenti.bin`. Il programma deve:

1. Chiedere l'indice da modificare e il nuovo valore della media.
2. Posizionarsi sul record desiderato.
3. Sovrascrivere esclusivamente quel record sul file aperto in modalità di aggiornamento "`r+b`".

--

```

1 #include <stdio.h>
2
3 typedef struct {
4     char nome[50];
5     int matricola;
6     float media;
7 } Studente;
8
9 int main(void) {
10     // Apriamo il file in lettura/scrittura senza distruggerlo (r+b)
11     FILE *fp = fopen("studenti.bin", "r+b");
12     if (fp == NULL) {
13         fprintf(stderr, "Errore: impossibile aprire il file. Eseguì
14         prima il salvataggio.\n");
15         return 1;
16     }
17     int indice = 0;
18     float nuova_media = 0.0;
19
20     printf("Inserisci l'indice dello studente da aggiornare (0, 1 o
21     2): ");
22     scanf("%d", &indice);
23     printf("Inserisci la nuova media: ");
24     scanf("%f", &nuova_media);
25
26     // 1. Spostiamo la testina sul record da aggiornare per
27     leggere prima
28     fseek(fp, indice * sizeof(Studente), SEEK_SET);
29     Studente s;
30     fread(&s, sizeof(Studente), 1, fp);
31
32     // 2. Modifichiamo il dato in memoria RAM
33     s.media = nuova_media;
34
35     // 3. Riposizioniamo la testina sullo stesso record prima di
36     sovrascriverlo
37     // (Nota: la fread ha fatto avanzare il cursore interno del
38     file)
39     fseek(fp, indice * sizeof(Studente), SEEK_SET);
40
41     // 4. Sovrascrittura mirata del blocco modificato
42     fwrite(&s, sizeof(Studente), 1, fp);
43     fclose(fp);
44     printf("Media aggiornata con successo su disco!\n");
45     return 0;
46 }

```

Spiegazione dell'algoritmo: L'apertura "r+b" consente di modificare porzioni del file preservando il resto del contenuto. Effettuiamo una prima fseek per leggere lo studente, aggiorniamo il campo media nella struct in RAM, dopodiché eseguiamo una seconda fseek per riposizionare indietro il cursore del file (poiché la precedente fread lo aveva fatto avanzare di sizeof(Studente) byte) ed effettuiamo la sovrascrittura tramite fwrite.

La Fuga dallo Stack (Memoria Dinamica e la nascita del Nodo)



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 14

La Fuga dallo Stack (Memoria Dinamica e la nascita del Nodo)

Nelle lezioni precedenti abbiamo visto come aggregare dati eterogenei usando le `struct` e come gestirli in blocchi fissi. Tuttavia, siamo sempre rimasti vincolati alla memoria *automatica* dello Stack, in cui le variabili nascono e muoiono rigidamente entro lo scope della funzione corrente.

In questa lezione compiremo un salto fondamentale: impareremo ad evadere dallo Stack allocando manualmente la memoria in un'area persistente controllata direttamente dal programmatore, lo **Heap**. Utilizzeremo le funzioni standard `malloc`, `calloc` e `free` per richiedere e rilasciare RAM, analizzeremo il funzionamento e i costi della riallocazione (`realloc`), e infine getteremo le basi per le strutture dati dinamiche definendo il **Nodo autoreferenziale**, l'unità fondamentale che ci consentirà di slegare i dati dalla contiguità fisica degli array.

14.1 Il Muro dello Stack e l'Ingresso nello Heap

14.1.1 Il problema irrisolvibile: Variabili Locali e Dangling Pointer

Si consideri il tentativo di scrivere una funzione che generi un array contenente i primi n numeri della successione di Fibonacci e lo restituisca al chiamante:

```
1  int* genera_fibonacci_sbagliato(int n) {  
2      // Array locale nello Stack (Variable Length Array - VLA)  
3      int seq[n];  
4  
5      seq[0] = 0;  
6      if (n > 1) seq[1] = 1;  
7  
8      for (int i = 2; i < n; i++) {  
9          seq[i] = seq[i-1] + seq[i-2];  
10     }  
11  
12     return seq; // Ritorno dell'indirizzo del primo elemento  
13 }
```

Compilando questo codice, il compilatore emette un avviso critico: `warning: function returns address of local variable [-Wreturn-local-addr]`

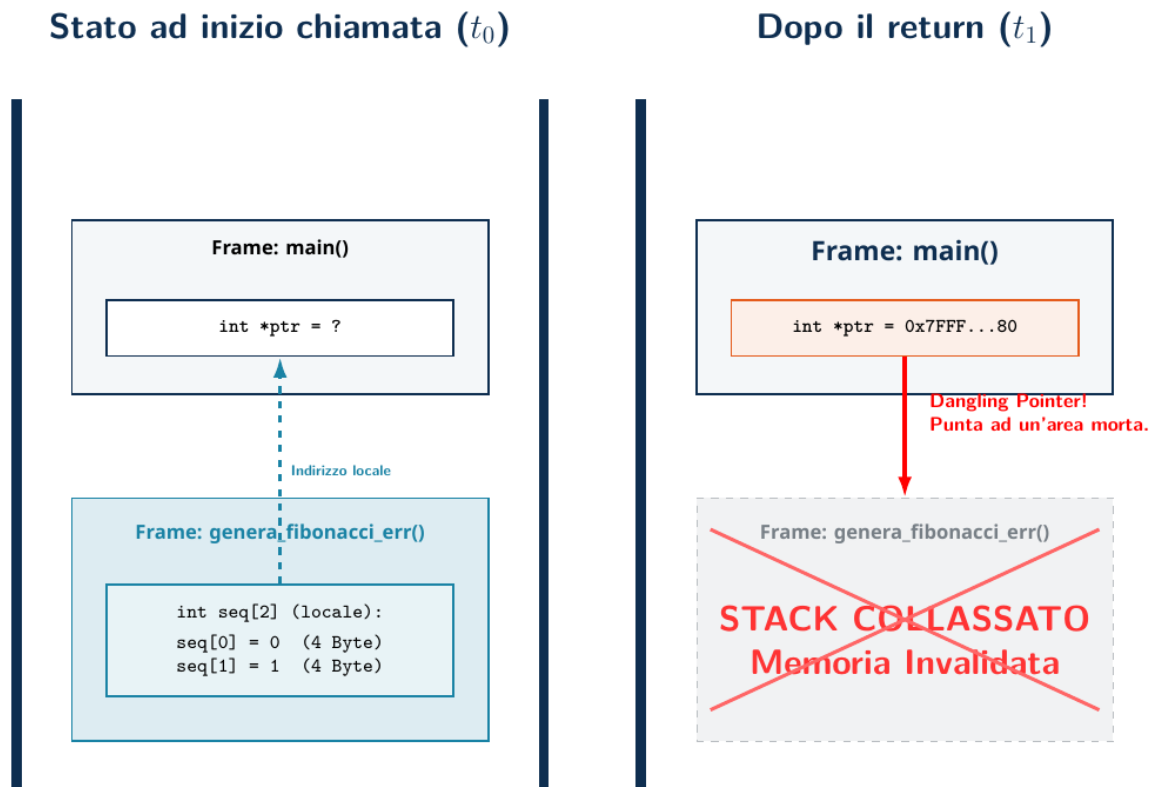


Figura 14.1: Collasso dello Stack

All'esecuzione, il programma subirà un crash immediato (Segmentation fault) o produrrà valori spazzatura.

La ragione fisica: L'array `seq` risiede nello Stack Frame di `genera_fibonacci_err()`. Al termine dell'esecuzione della funzione (`return`), il suo intero frame viene rimosso dallo Stack e invalidato dal sistema operativo. L'indirizzo restituito punta ad un'area non più riservata (un **Dangling Pointer**), pronta ad essere sovrascritta da altre chiamate a funzione.

Attenzione

Attenzione Tutte le variabili create nello Stack nascono e muoiono con lo scope della funzione in cui sono dichiarate. Non è possibile restituire puntatori a variabili locali o array locali dallo stack.

14.1.2 La Fuga: La Memoria HEAP

Per superare la logica LIFO (Last-In, First-Out) dello Stack, i sistemi operativi offrono l'area di memoria **Heap** (Mucchio):

- **Ampiezza:** È limitata solo dalla RAM fisica complessiva del sistema.
- **Persistenza:** La memoria allocata nello Heap sopravvive alla chiusura delle funzioni e rimane attiva fino a quando il programmatore non ne richiede esplicitamente la distruzione, o fino al termine dell'intero processo.

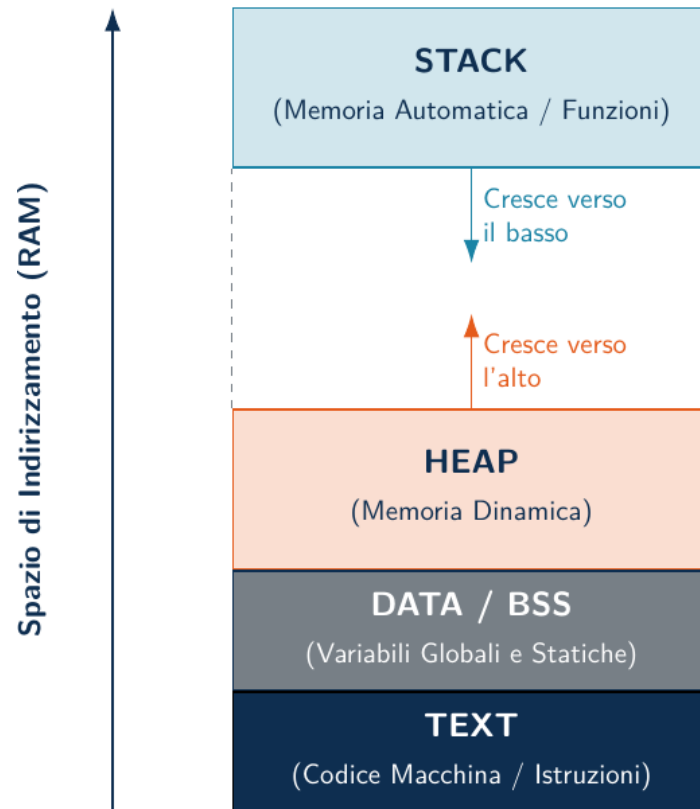


Figura 14.2: Stack e Heap

- **Manuale:** La gestione è interamente affidata allo sviluppatore.

14.1.3 Le Chiavi dello Heap: `malloc` e `free`

La libreria standard `<stdlib.h>` fornisce gli strumenti per dialogare con lo Heap.

Allocazione con `malloc`

La funzione `malloc` (Memory Allocation) richiede al sistema operativo un blocco continuo di byte:

```
1 void* malloc(size_t size);
```

- Riceve come parametro il numero esatto di byte da allocare.
- Restituisce un puntatore a tipo generico (`void*`) che punta all'inizio del blocco allocato nello Heap.
- Se l'allocazione fallisce (ad es. per mancanza di RAM), restituisce `NULL`.

Riscriviamo la generazione di Fibonacci in sicurezza:

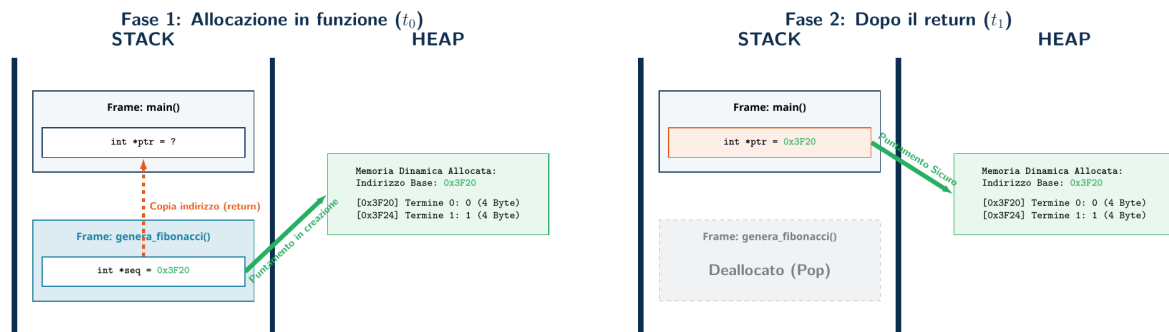


Figura 14.3: Ciclo di Vita Heap

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int* genera_fibonacci(int n) {
5      // Chiediamo spazio nell'Heap per n interi
6      int *seq = (int*) malloc(n * sizeof(int));
7
8      if (seq == NULL) {
9          fprintf(stderr, "Errore: Memoria insufficiente!\n");
10         return NULL;
11     }
12
13     seq[0] = 0;
14     if (n > 1) seq[1] = 1;
15
16     for (int i = 2; i < n; i++) {
17         seq[i] = seq[i-1] + seq[i-2];
18     }
19
20     return seq; // Perfettamente sicuro: l'Heap non collassa al return!
21 }

```

L'immagine che segue mostra come si può usare lo heap per costruire un nuovo array dentro una funzione e restituire un puntatore al chiamante:

Rilascio della memoria con free

Poiché la memoria Heap non viene pulita automaticamente, dimenticare di liberarla genera un **Memory Leak** (perdita di memoria). La memoria occupata rimane bloccata e indisponibile. Per liberare un blocco di memoria si usa `free`:

```

1 void free(void* ptr);

```

Dopo aver liberato un puntatore, è ottima norma impostarlo a `NULL` per evitare di riutilizzarlo accidentalmente (Dangling Pointer).

```

1 int main(void) {
2     int n = 8;
3     int *fib = genera_fibonacci(n);

```

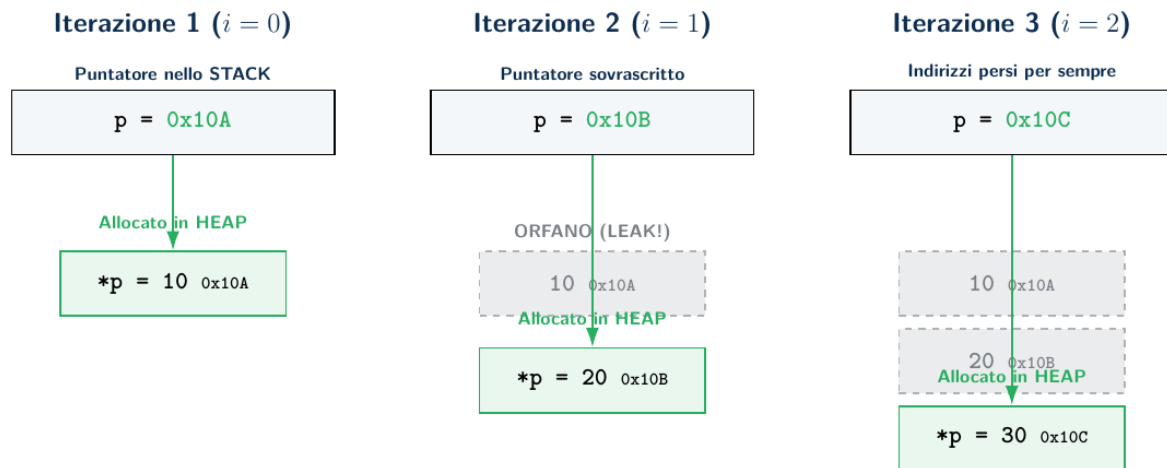


Figura 14.4: Memory Leak

```

4
5  if (fib != NULL) {
6      for (int i = 0; i < n; i++) {
7          printf("%d ", fib[i]);
8      }
9      printf("\n");
10
11     free(fib); // Riconsegniamo i byte allo Heap
12     fib = NULL; // Annulliamo il puntatore per sicurezza
13 }
14 return 0;
15 }

```

L'immagine che segue mostra un esempio di memory leak:

14.2 Array Dinamici e Ridimensionamento

14.2.1 Sintassi di Accesso negli Array Dinamici

Una volta allocato un array dinamico e assegnato l'indirizzo a un puntatore, l'accesso avviene con la medesima sintassi degli array statici:

```

1  int *arr = (int*) malloc(5 * sizeof(int));
2  arr[0] = 10;
3  *(arr + 1) = 20; // Equivalente a arr[1] = 20;

```

L'operatore parentesi quadre `[]` compie implicitamente l'aritmetica dei puntatori ed esegue la dereferenziazione.

14.2.2 `calloc`: Allocazione con Inizializzazione a Zero

A differenza di `malloc`, che lascia intatto il contenuto preesistente della memoria (spazzatura), la funzione `calloc` pulisce a zero tutti i bit allocati:

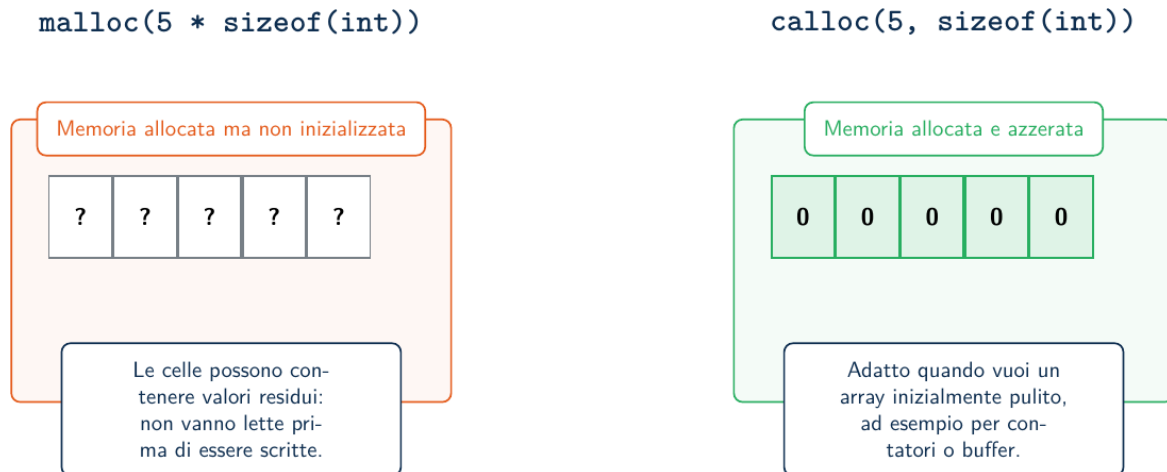


Figura 14.5: malloc vs calloc

```
1 void* calloc(size_t nmemb, size_t size);
```

```
1 // Alloca 10 interi e li inizializza tutti a 0
2 int *vettore = (int*) calloc(10, sizeof(int));
```

Il diagramma seguente mostra le differenze fisiche nell'inizializzazione della memoria tra malloc e calloc:

14.2.3 Ridimensionare la memoria: realloc

Se la dimensione originaria di un array dinamico si rivela insufficiente, possiamo modificarne l'estensione tramite realloc:

```
1 void* realloc(void* ptr, size_t size);
```

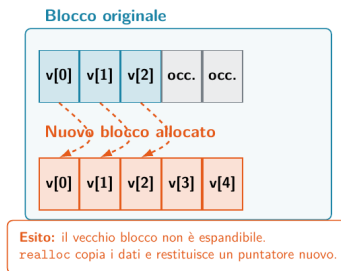
Meccanismo di funzionamento:

1. **Espansione in situ:** Se lo spazio contiguo immediatamente successivo nello Heap è libero, la realloc estende semplicemente il blocco originale senza spostarlo.
2. **Trasloco:** Se lo spazio adiacente è occupato, la realloc cerca una nuova area capiente altrove nello Heap, copia l'intero contenuto, dealloca il vecchio spazio in automatico e restituisce il nuovo indirizzo.
3. **Fallimento:** Se non c'è RAM sufficiente, restituisce NULL lasciando il puntatore originario valido e intatto.

Scenario A: Espansione in avanti



Scenario B: Copia su nuovo blocco



Scenario C: Fallimento

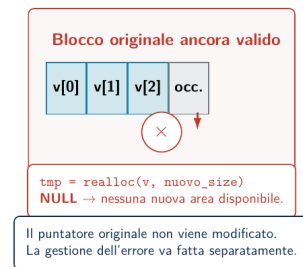


Figura 14.6: Scenari realloc

Nota

Nota Per evitare perdite di memoria in caso di fallimento della `realloc`, non bisogna mai sovrascrivere direttamente il puntatore originale con il risultato della funzione senza prima controllare se è valido (NULL).

```
1 // Corretto schema d'uso di realloc
2 int *nuovo = (int*) realloc(fib, (n + 1) * sizeof(int));
3 if (nuovo != NULL) {
4     fib = nuovo;
5 } else {
6     // Gestione errore: fib contiene ancora i dati precedenti!
7 }
```

Il diagramma seguente illustra i tre diversi scenari che si possono verificare durante una chiamata a `realloc`:

14.3 Verso le Strutture Dinamiche: Il Nodo

14.3.1 Superare il vincolo della contiguità

Gli array in C devono essere allineati in celle di memoria contigue. Ciò rende costoso l'inserimento o la riallocazione in mezzo ai dati. L'alternativa consiste nel disporre i dati sparsi in zone distinte dello Heap, legandoli tra loro tramite puntatori, in una caccia al tesoro logica.

14.3.2 Il Nodo Autoreferenziale

L'elemento base di questa catena è la struct autoreferenziale, detta **Nodo**:

```
1 typedef struct Nodo {
2     int dato; // Il valore utile (Payload)
3     struct Nodo *prossimo; // Il puntatore al prossimo nodo
4 } Nodo;
```

Attenzione

Attenzione Dentro la definizione della struct non è possibile usare l'alias `Nodo` in quanto non è ancora stato registrato dal compilatore. Bisogna usare obbligatoriamente la nomenclatura formale completa `struct Nodo *prossimo`.

14.3.3 Collegamento manuale di nodi

Il seguente codice mostra come allocare e collegare manualmente tre nodi nello Heap:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  int main(void) {
10     // 1. Allocazione dei nodi sparsi nello Heap
11     Nodo *testa = (Nodo*) malloc(sizeof(Nodo));
12     Nodo *secondo = (Nodo*) malloc(sizeof(Nodo));
13     Nodo *terzo = (Nodo*) malloc(sizeof(Nodo));
14
15     // 2. Assegnamento dei dati
16     testa->dato = 10;
17     secondo->dato = 20;
18     terzo->dato = 30;
19
20     // 3. Collegamento
21     testa->prossimo = secondo;
22     secondo->prossimo = terzo;
23     terzo->prossimo = NULL; // Fine della catena
24
25     // Navigazione
26     printf("Secondo dato: %d\n", testa->prossimo->dato); // 20
27
28     // Liberazione manuale
29     free(testa);
30     free(secondo);
31     free(terzo);
32
33     return 0;
34 }
```

Il diagramma seguente illustra la disposizione in memoria dei nodi nello Heap e il relativo collegamento tramite puntatori:

14.4 Autovalutazione ed Esercizi

Heap: nodi sparsi e collegati

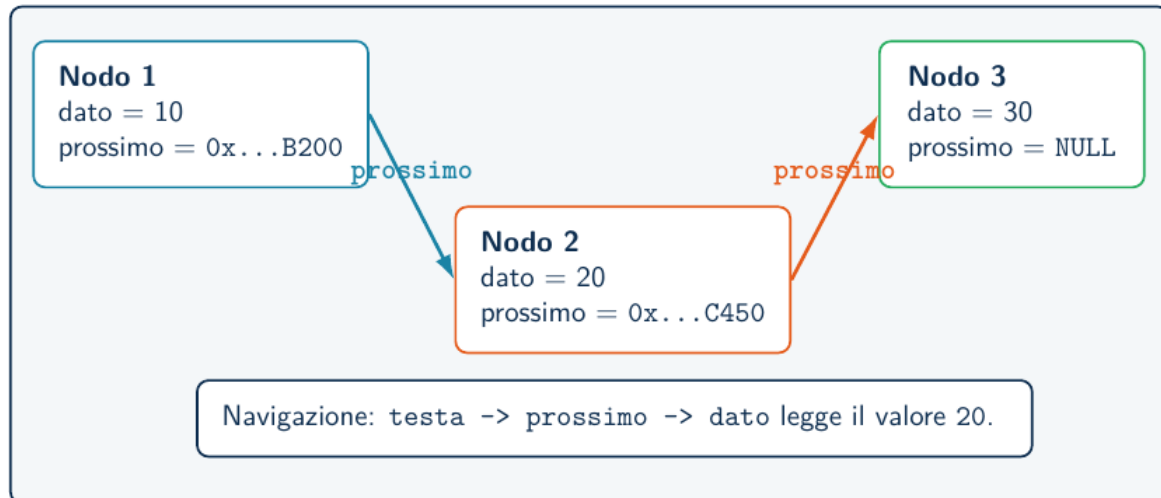


Figura 14.7: Collegamento dei Nodi

14.4.1 Test a Risposta Multipla (10 Quiz)

Quiz 1

Domanda

1. Cosa succede se una funzione restituisce l'indirizzo di una variabile locale allocata nello Stack?

- a) La variabile viene promossa automaticamente nello Heap.
- b) Si ottiene un dangling pointer che punta a una zona di memoria invalidata dal return.
- c) Il compilatore rifiuta di compilare bloccando l'esecuzione con errore bloccante.
- d) La variabile sopravvive fino al termine del main.

Soluzione
Risposta: b. Le variabili locali nello Stack cessano di esistere al return; il puntatore diventa dangling.

Quiz 2

Domanda

2. Quale delle seguenti affermazioni descrive correttamente la memoria Heap?

- a) È ordinata secondo la logica LIFO.
- b) Viene gestita automaticamente dal compilatore senza intervento umano.

- c) Consente allocazioni persistenti a vita manuale, fino alla chiamata di `free`.
- d) Ha dimensioni inferiori a quelle dello Stack di sistema.

Soluzione
Risposta: c. Lo Heap è governato interamente in modo manuale dal programmatore.

Quiz 3

Domanda

3. Cosa restituisce `malloc` in caso di fallimento dell'allocazione?

- a) Un puntatore a `0x0001`.
- b) Un valore casuale.
- c) `NULL`.
- d) Genera un'eccezione di runtime interrompendo il programma.

Soluzione
Risposta: c. Restituisce `NULL`, motivo per cui va sempre controllata.

Quiz 4

Domanda

4. Qual è la principale differenza tra `malloc` e `calloc`?

- a) `calloc` alloca solo memoria nello Stack.
- b) `calloc` alloca la memoria e azzerà il valore di tutti i bit.
- c) `malloc` accetta due parametri anziché uno.
- d) `calloc` non può fallire.

Soluzione
Risposta: b. `calloc` inizializza a zero, `malloc` lascia spazzatura.

Quiz 5

Domanda

5. Cosa si intende per "Memory Leak"?

- a) La perdita fisica del computer contenente i file sorgente.
- b) Una fuga di dati sensibili a causa di un puntatore non protetto.

- c) Memoria Heap allocata che non viene liberata con `free`, rimanendo inutilizzabile.
- d) Il collasso dello Stack Frame per ricorsione infinita.

Soluzione
Risposta: c. È l'accumulo di memoria allocata e mai rilasciata.

Quiz 6

Domanda

6. Come si comporta `realloc` se non trova spazio adiacente per allargare il blocco?

- a) Restituisce `NULL` e distrugge i vecchi dati.
- b) Sposta l'intero blocco in una nuova posizione nello Heap e copia i dati.
- c) Converte l'array in una lista concatenata in modo trasparente.
- d) Sospende l'esecuzione del programma fino a quando non si libera spazio.

Soluzione
Risposta: b. Eseguendo un "trasloco" dei dati in una nuova area Heap.

Quiz 7

Domanda

7. Perché nello schema `realloc` non si deve riassegnare direttamente il puntatore originale?

- a) Perché l'assegnamento diretto causerebbe un errore sintattico.
- b) Perché se la `realloc` fallisce e restituisce `NULL`, perderemmo l'indirizzo originale creando un memory leak.
- c) Perché il puntatore originale verrebbe invalidato in automatico dal compilatore.
- d) Perché la `realloc` richiede un cast esplicito a `void**`.

Soluzione
Risposta: b. Sovrascrivere con `NULL` rende inaccessibile il vecchio blocco.

Quiz 8

Domanda

8. Qual è la sintassi corretta per un puntatore autoreferenziale in una struct chiamata `Nodo`?

- a) `Nodo *prossimo;` all'interno della struct senza alias definito.
- b) `struct Nodo *prossimo;`
- c) `void *prossimo;` castato sempre a `struct Nodo`.
- d) `struct prossimo *Nodo;`

Soluzione
Risposta: b. Poiché l'alias `Nodo` non è ancora registrato, è necessario usare `struct`
`Nodo *`.

Quiz 9

Domanda

9. Che valore si assegna tipicamente al campo `prossimo` dell'ultimo nodo di una lista?

- a) `0x0001`.
- b) `EOF`.
- c) `NULL`.
- d) L'indirizzo del nodo di testa (sempre).

Soluzione
Risposta: c. `NULL` funge da terminatore.

Quiz 10

Domanda

10. A quale delle seguenti operazioni equivale sintatticamente `*(ptr + i)`?

- 1 - a) `'ptr->i'`
- 2
- 3 - b) `'ptr[i]'`
- 4
- 5 - c) `'&ptr[i]'`
- 6
- 7 - d) `'ptr.i'`

Soluzione
Risposta: b. L'accesso con parentesi quadre è pura aritmetica e dereferenziazione di puntatori.

14.4.2 Esercizi di Programmazione Progressivi

Esercizio 1: Allocazione di un Intero Singolo

Scrivere un programma che allochi dinamicamente memoria per un singolo intero nello Heap, vi assegni il valore 42, lo stampi, liberi la memoria e azzeri il puntatore.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int *p = malloc(sizeof(int));
7      if (p == NULL)
8          return 1;
9      *p = 42;
10     printf("Valore nello Heap: %d\n", *p);
11     free(p);
12     p = NULL;
13     return 0;
14 }

```

Soluzione

!

Esercizio 2: Vettore Dinamico Semplice

Scrivere una funzione `double* crea_vettore(int n)` che allochi un array di `n` double tramite `calloc`, inserisca in ciascuna cella il proprio indice e restituisca il puntatore. Testare la funzione nel main e liberare la memoria.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  double* crea_vettore(int n)
5  {
6      double *arr = calloc(n, sizeof(double));
7      if (arr == NULL)
8          return NULL;
9      for (int i = 0; i < n; i++)
10         arr[i] = i;
11     return arr;
12 }

```

Soluzione

!

Esercizio 3: Stringa Dinamica da Input

Scrivere un programma che legga da tastiera un intero `lunghezza` e allochi dinamicamente una stringa (`char*`) della dimensione esatta (incluso il terminatore `\0`). Leggere la stringa con `scanf` e stamparla al contrario.

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6     int len;
7     printf("Inserisci la lunghezza massima: ");
8     if (scanf("%d", &len) != 1)
9         return 1;
10    char *s = (char*) malloc((len + 1) * sizeof(char));
11    if (s == NULL)
12        return 1;
13    printf("Inserisci la stringa: ");
14    scanf("%s", s);
15    int reale = strlen(s);
16    for (int i = reale - 1; i >= 0; i--)
17        putchar(s[i]);
18    putchar("\n");
19    free(s);
20    return 0;
21 }

```

Soluzione

!

Esercizio 4: Prevenzione di Perdite con Malloc

Creare una funzione che allochi un array di 1000 interi. Simulare un fallimento inserendo intenzionalmente un ciclo che non libera la memoria. Aggiungere le istruzioni di rilascio necessarie per mantenere la stabilità del sistema ed evitare il leak.

```

// Rilascio corretto free(temp);
int main(void) for (int i = 0; i < 10000; i++) alloca_era(lascia()); // Senza free provocherebbe leak continuo printf("E
void alloca_era(lascia(void) int * temp = (int*) malloc(1000 * sizeof(int)); if(temp != NULL) // Operazioni fini
bi include <stdio.h> include <stdlib.h>

```

Soluzione

!

Esercizio 5: Espansione di un Buffer con Realloc

Creare un array dinamico di 3 interi, popolarlo con i valori 10, 20, 30. Ridimensionarlo a 5 elementi usando realloc e inserire nelle nuove posizioni 40, 50. Stampare l'array finale.

```

bi include <stdio.h> include <stdlib.h>
int main(void) int *arr = (int*) malloc(3 * sizeof(int)); if (arr == NULL) return 1;
arr[0] = 10; arr[1] = 20; arr[2] = 30;
int *nuovo = (int*) realloc(arr, 5 * sizeof(int)); if (nuovo == NULL) free(arr); return 1; arr =
nuovo; arr[3] = 40; arr[4] = 50;
for (int i = 0; i < 5; i++) printf("%i", arr[i]);
free(arr); return 0;

```

Soluzione

!

Esercizio 6: Stringa Dinamica Concatenata

Scrivere una funzione char* concatena(const char *s1, const char *s2) che allochi dinamicamente lo spazio strettamente necessario a ospitare la concatenazione di due stringhe, la esegua e restituisca il puntatore al main.

```

bi include <stdio.h> include <string.h>
char* concatena(const char *s1, const char *s2) int l1 = strlen(s1); int l2 = strlen(s2); char
*res = (char*) malloc((l1 + l2 + 1) * sizeof(char)); if (res == NULL) return NULL; strcpy(res, s1);
strcat(res, s2); return res;
int main(void) char *c = concatena("Hello, ", "World!"); if (c != NULL) printf("free(c); c = NULL;
return 0;

```

Soluzione

!

Esercizio 7: Copia Dinamica di una Struct

Data una struct typedef struct \{\} char modello[20]; float prezzo; \{\} Auto;, scrivere una funzione Auto* duplica_auto(const Auto *originale) che al-

lochi nello Heap una copia identica della struct passata per riferimento e ne restituisca l'indirizzo.

```

        NULL); printf("Copia : free(d); return 0;
        int main(void) { Fiat Panda", 14500.0f, Auto * d = duplica
        Auto * duplica (const Auto * originale) { Auto * copia = (Auto *) malloc(sizeof(Auto)); if(copia == NULL) r
        typedef struct char modello[20]; float prezzo; Auto;
        b) include <stdio.h> include <stdlib.h> include <string.h>
        Soluzione

```

!

Esercizio 8: Creazione di una Lista Dinamica di 3 Nodi

Scrivere un programma che definisca il tipo Nodo e popoli dinamicamente una catena di 3 nodi con valori inseriti dall'utente a runtime. Stampare la lista scorrendola tramite un puntatore temporaneo.

```

        free(testa); free(secondo); free(terzo); return 0;
        printf("curr = curr->prossimo; printf("NULL");
        // Scorrimento pulito con puntatore temporaneo Nodo * curr = testa; while (curr != NULL)
        testa->prossimo = secondo; secondo->prossimo = terzo; terzo->prossimo = NULL;
        printf("Inserisci tre interi:"); scanf(
        if (testa == NULL || secondo == NULL || terzo == NULL) return 1;
        malloc(sizeof(Nodo)); Nodo * terzo = (Nodo *) malloc(sizeof(Nodo));
        int main(void) { Nodo * testa = (Nodo *) malloc(sizeof(Nodo)); Nodo * secondo = (Nodo *)
        typedef struct Nodo int valore; struct Nodo * prossimo; Nodo;
        b) include <stdio.h> include <stdlib.h>
        Soluzione

```

!

Esercizio 9: Matrice Lineare Dinamica (Row-Major Order)

Scrivere un programma che allochi un array monodimensionale nello Heap per simulare una griglia di righe e colonne (matrice linearizzata). Leggere righe R e colonne C, popolare la matrice con il prodotto degli indici $i * j$ e stamparla in forma tabellare.

```

        free(mat); return 0;
        for (int i = 0; i < R; i++) for (int j = 0; j < C; j++) printf("%d",
        for (int i = 0; i < R; i++) for (int j = 0; j < C; j++) mat[i * C + j] = i * j; // Formula Row-Major Order
        = (int *) malloc(R * C * sizeof(int)); if (mat == NULL) return 1;
        int main(void) { int R = 3, C = 4; // Linearizzazione: allocazione di R * C celle contigue int * mat
        b) include <stdio.h> include <stdlib.h>
        Soluzione

```

!

Esercizio 10: Inserimento in Testa (Manuale)

Dato il puntatore iniziale `Nodo *testa` che punta a una catena esistente di 2 nodi (`10 -> 20 -> NULL`), implementare manualmente l'inserimento di un nuovo nodo di valore 5 in testa, aggiornando correttamente i puntatori. Stampare la nuova lista partendo da `testa`.

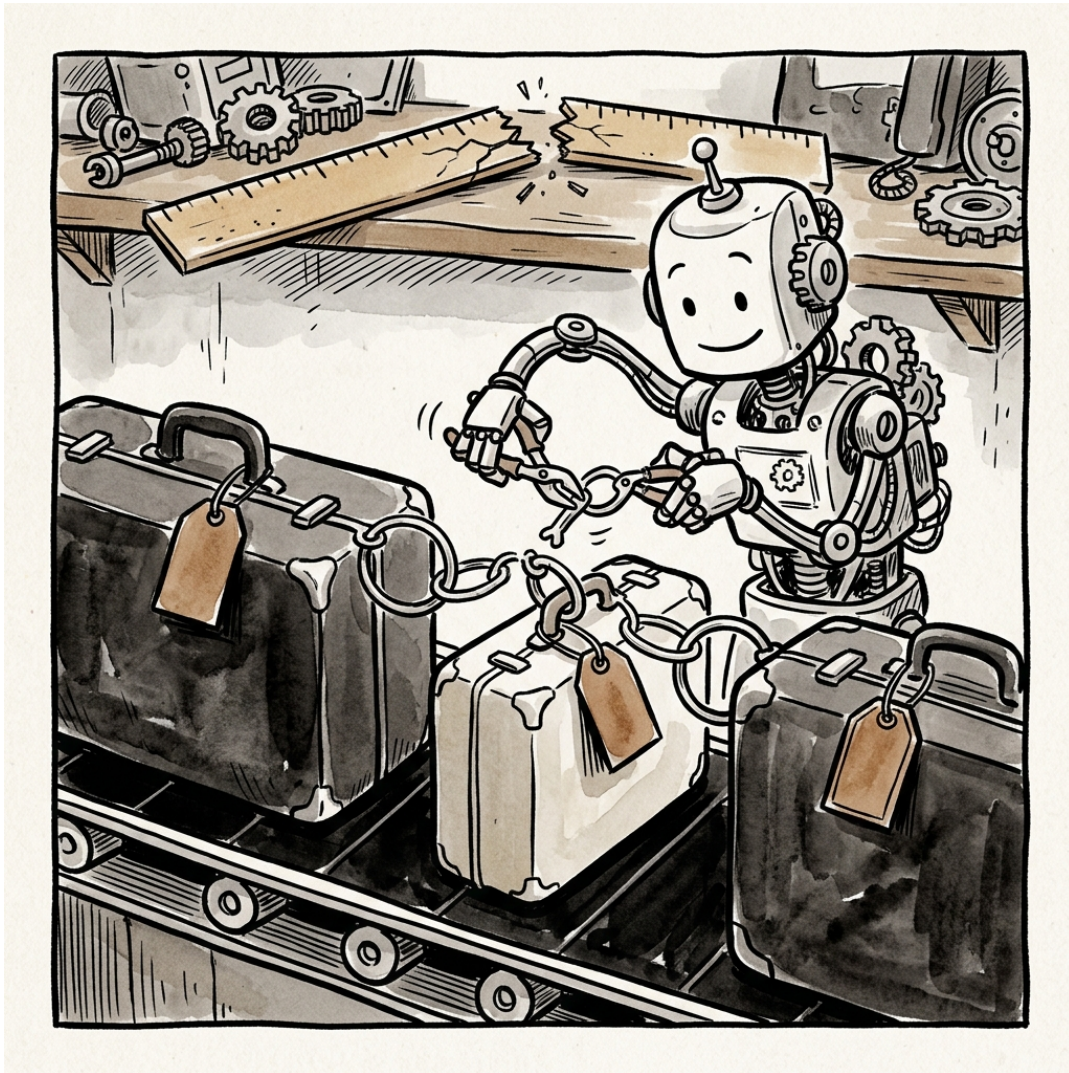
```

bi nclude<stdio.h> include<stdlib.h>
typedef struct Nodo {int valore; struct Nodo *prossimo;
int main(void) // Lista di partenza: 10 -> 20 -> NULL
Nodo *n1 = (Nodo*) malloc(sizeof(Nodo));
Nodo *n2 = (Nodo*) malloc(sizeof(Nodo)); if (n1 == NULL || n2 == NULL) return 1; n1->valore = 10; n2->valore = 20; n1->prossimo = n2; n2->prossimo = NULL;
Nodo *testa = n1;
// NUOVO NODO DA INSERIRE IN TESTA
Nodo *nuovo = (Nodo*) malloc(sizeof(Nodo)); if
(nuovo == NULL) return 1; nuovo->valore = 5;
// INSERIMENTO: il nuovo nodo punta alla vecchia testa nuovo->prossimo = testa; // La testa
si sposta sul nuovo nodo testa = nuovo;
// Stampa finale per validazione
Nodo *temp = testa; while (temp != NULL) printf("temp =
temp->prossimo; printf("NULL");
// Liberazione di tutta la memoria free(n2); free(n1); free(nuovo);
return 0;

```

Soluzione

Liste Concatenate Semplici



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 15

Liste Concatenate Semplici

Fino ad oggi abbiamo memorizzato collezioni di elementi contigui in memoria fisica (RAM) attraverso gli array statici e dinamici. Sebbene gli array garantiscano tempi di accesso immediati in tempo costante grazie al calcolo dell'offset dall'indice di partenza, essi presentano rigidi limiti strutturali:

1. **Costo di ridimensionamento:** Se lo spazio allocato nello Heap tramite `malloc` o `realloc` si esaurisce, l'ingrandimento dell'array può comportare la ricerca di una nuova area di memoria contigua sufficientemente capiente, costringendo la CPU a copiare interamente i dati esistenti ($O(N)$).
2. **Costo di inserimento/eliminazione in posizione intermedia:** Inserire o rimuovere un valore all'indice 0 richiede lo scorrimento (*shifting*) di tutti gli elementi successivi verso destra o verso sinistra, con una complessità pari a $O(N)$.

La **Lista Concatenata Semplice** (Singly Linked List) supera questi limiti rinunciando alla contiguità fisica degli elementi.

15.1 La Struttura Logica della Lista

In una lista concatenata, gli elementi (chiamati **Nodi**) sono distribuiti liberamente all'interno dello Heap. Essi sono logicamente collegati tramite indirizzi di memoria (puntatori).

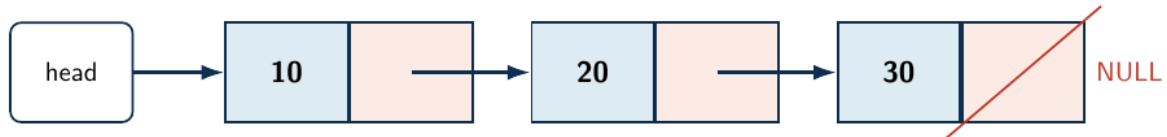
15.1.1 L'Anatomia di un Nodo

Ogni nodo è un record (`struct`) costituito da due campi fondamentali:

1. **Payload (Dato):** Le informazioni effettivamente memorizzate (ad esempio, un intero `int`, un carattere o una struttura nidificata).
2. **Puntatore al Successore (next):** Un puntatore contenente l'indirizzo di memoria del nodo successivo nella sequenza.

Per gestire una lista concatenata, il programmatore deve mantenere una variabile locale nello Stack chiamata **Puntatore di Testa (head)**, la quale contiene l'indirizzo di memoria del primo nodo.

Lista con tre nodi



Ogni cella “dato” contiene il valore; ogni cella “next” contiene l'indirizzo del nodo successivo.

Figura 15.1: Struttura di una lista concatenata

15.2 Rappresentazione in C ed Ispezione

Definiamo il nodo in C mediante una struttura auto-referenziale (una struct che contiene un puntatore a un'altra istanza della stessa struct):

```

1 typedef struct Nodo {
2     int dato;
3     struct Nodo *prossimo;
4 } Nodo;
  
```

15.2.1 Funzione di Allocazione Sicura

Per evitare di scrivere ripetutamente codice di allocazione nello Heap con gestione degli errori, implementiamo una funzione helper per creare nodi isolati:

```

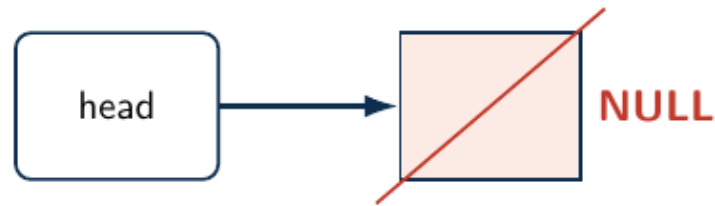
1 Nodo* crea_nodo(int valore) {
2     Nodo *nuovo = (Nodo*) malloc(sizeof(Nodo));
3     if (nuovo == NULL) {
4         fprintf(stderr, "Errore fatale: memoria Heap esaurita.\n");
5         exit(EXIT_FAILURE);
6     }
7     nuovo->dato = valore;
8     nuovo->prossimo = NULL; // Nasce scollegato
9     return nuovo;
10 }
  
```

Appena creata, la lista è composta solo della testa ed è dunque vuota, come mostrato nell'immagine seguente:

Possiamo verificare se una lista è vuota mediante la seguente funzione:

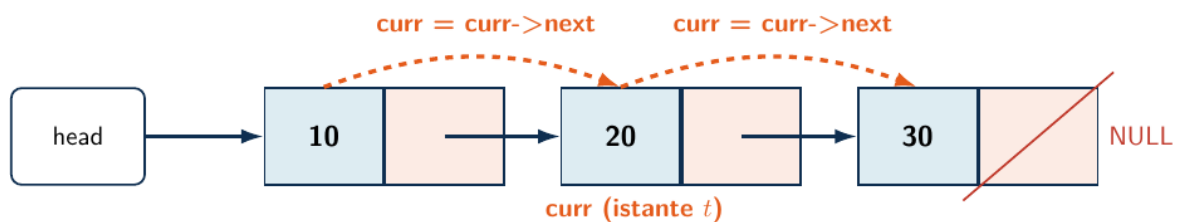
```

1 bool is_empty(Nodo *testa) {
2     return testa == NULL;
3 }
  
```



head punta a NULL: nessun nodo esiste, la lista non ha elementi.

Figura 15.2: Lista vuota



Il ciclo termina quando curr diventa NULL: tutti i nodi sono stati visitati.

Figura 15.3: Attraversamento della lista

15.2.2 Scorrimento Sequenziale e Stampa

Per scorrere la lista, non possiamo ricorrere ad indici numerici diretti. Dobbiamo usare un puntatore ausiliario (convenzionalmente chiamato *corrente* o *curr*) posizionato inizialmente sulla testa, facendolo avanzare progressivamente aggiornando il suo indirizzo con il valore salvato nel campo *prossimo*.

```

1 void stampa_lista(Nodo *testa) {
2     Nodo *corrente = testa;
3     printf("Testa -> ");
4     while (corrente != NULL) {
5         printf("[%d] -> ", corrente->dato);
6         corrente = corrente->prossimo; // Passo di avanzamento
7     }
8     printf("NULL\n");
9 }

```

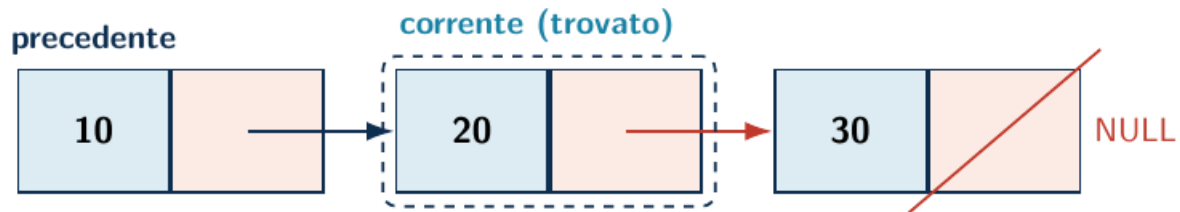
15.2.3 Ricerca di un Elemento

La ricerca in una lista concatenata non ordinata è intrinsecamente lineare e richiede di scorrere i nodi uno ad uno:

```

1 bool cerca_elemento(Nodo *testa, int chiave, Nodo **nodo_trovato) {
2     Nodo *corrente = testa;

```



Il ciclo di ricerca si ferma su `corrente->dato == 20`.

Figura 15.4: Ricerca del nodo

```

3  while (corrente != NULL) {
4      if (corrente->dato == chiave) {
5          if (nodo_trovato != NULL) {
6              *nodo_trovato = corrente;
7          }
8          return true;
9      }
10     corrente = corrente->prossimo;
11 }
12 if (nodo_trovato != NULL) {
13     *nodo_trovato = NULL;
14 }
15 return false;
16 }

```

Quando non si conosce già la posizione del nodo, si scorre la lista con due puntatori, *precedente* e *corrente*, finché `corrente->dato` non corrisponde al valore cercato; poi si applica la stessa logica di detach.

15.3 La Chirurgia dei Puntatori: Inserimenti

L'inserimento di elementi in una lista non richiede mai lo spostamento fisico dei nodi in memoria, bensì la riconfigurazione locale delle connessioni dei puntatori.

15.3.1 Inserimento in Testa

Inserire in testa è un'operazione che richiede tempo costante:

1. Si alloca un nuovo nodo.
2. Si fa puntare il campo *prossimo* del nuovo nodo alla vecchia testa.
3. Si aggiorna il puntatore di testa affinché contenga l'indirizzo del nuovo nodo.

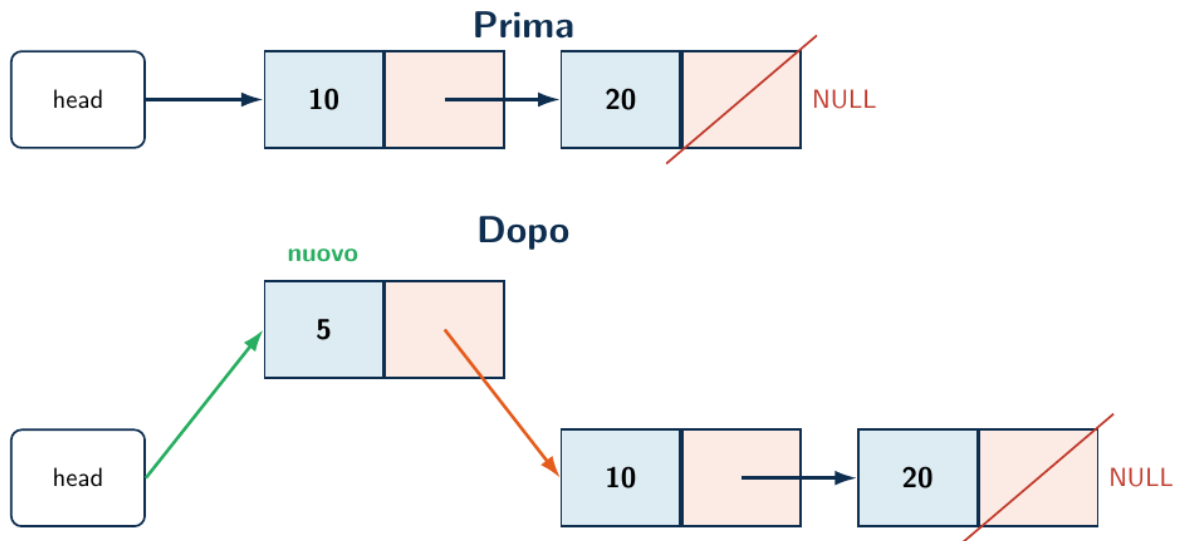


Figura 15.5: Inserimento in testa

```

1  Nodo* inserisci_testa(Nodo *testa, int valore) {
2      Nodo *nuovo = crea_nodo(valore);
3      nuovo->prossimo = testa;
4      return nuovo; // Diventa la nuova testa
5  }

```

Nelle figure sottostanti viene mostrato il passaggio dall'allocazione isolata del nuovo nodo all'aggancio definitivo.

Il codice sopra funziona anche nel caso speciale in cui inseriamo un elemento in una lista vuota:

15.3.2 Inserimento in Coda

Per inserire un elemento in fondo alla lista, è necessario individuare l'ultimo nodo (ovvero il nodo che punta a NULL), modificando il suo campo `prossimo` affinché punti al nuovo nodo.

- Se la lista è vuota (`head == NULL`), il nuovo nodo diventa direttamente la testa.
- Se la lista contiene elementi, si effettua una scansione lineare per posizionarsi sull'ultimo elemento.

```

1  Nodo* inserisci_coda(Nodo *testa, int valore) {
2      Nodo *nuovo = crea_nodo(valore);
3      if (testa == NULL) {
4          return nuovo;
5      }
6      Nodo *corrente = testa;
7      while (corrente->prossimo != NULL) {
8          corrente = corrente->prossimo;
9      }
10     corrente->prossimo = nuovo;

```

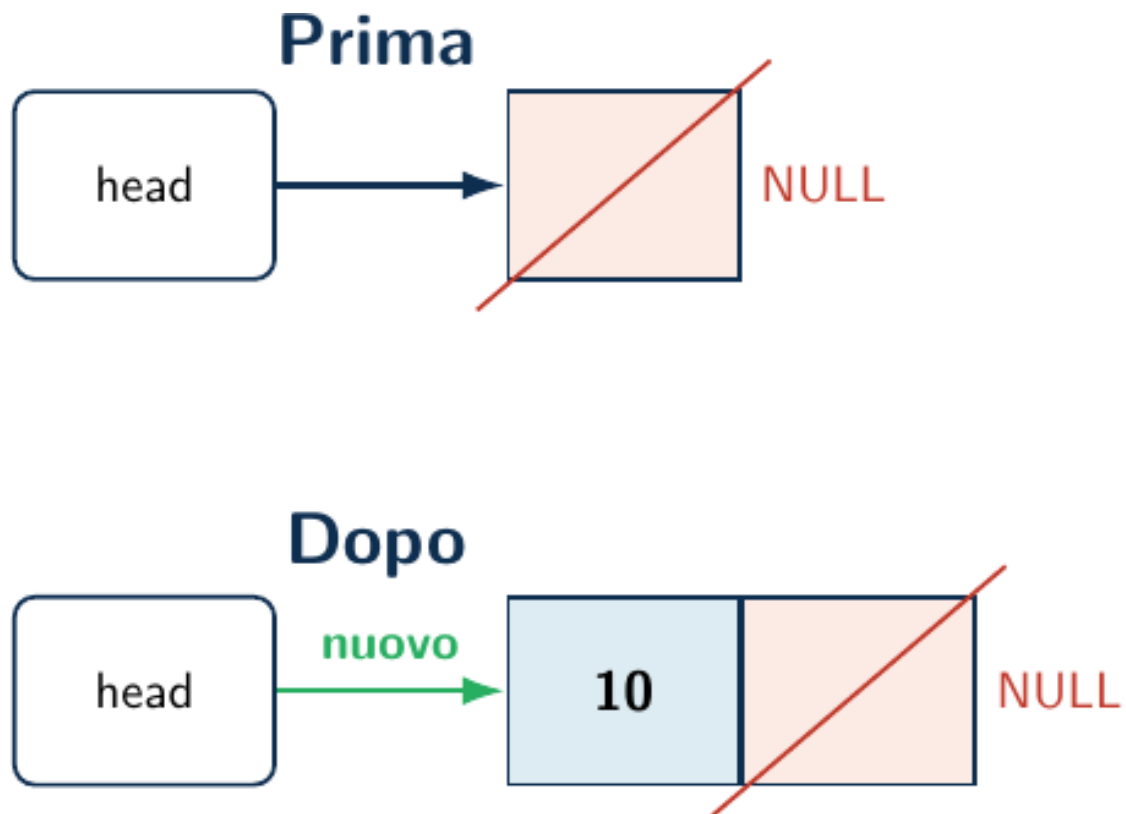


Figura 15.6: Inserimento in lista vuota

```

11  return testa;
12  }

```

15.3.3 Inserimento Ordinato

L'inserimento ordinato in una lista crescente richiede di trovare la posizione esatta in mezzo a due nodi esistenti. Poiché la lista semplice è a senso unico, una volta superato il punto di inserimento non è possibile tornare indietro per aggiornare il puntatore del nodo che precede. Per risolvere questo problema, si utilizza la **tecnica dei due puntatori sfalsati** (precedente e corrente).

```

1  Nodo* inserisci_ordinato(Nodo *testa, int valore) {
2      Nodo *nuovo = crea_nodo(valore);
3
4      // Caso 1: Lista vuota o inserimento in testa
5      if (testa == NULL || valore < testa->dato) {
6          nuovo->prossimo = testa;
7          return nuovo;
8      }
9
10     // Caso 2: Ricerca della posizione intermedia
11     Nodo *precedente = testa;
12     Nodo *corrente = testa->prossimo;
13

```

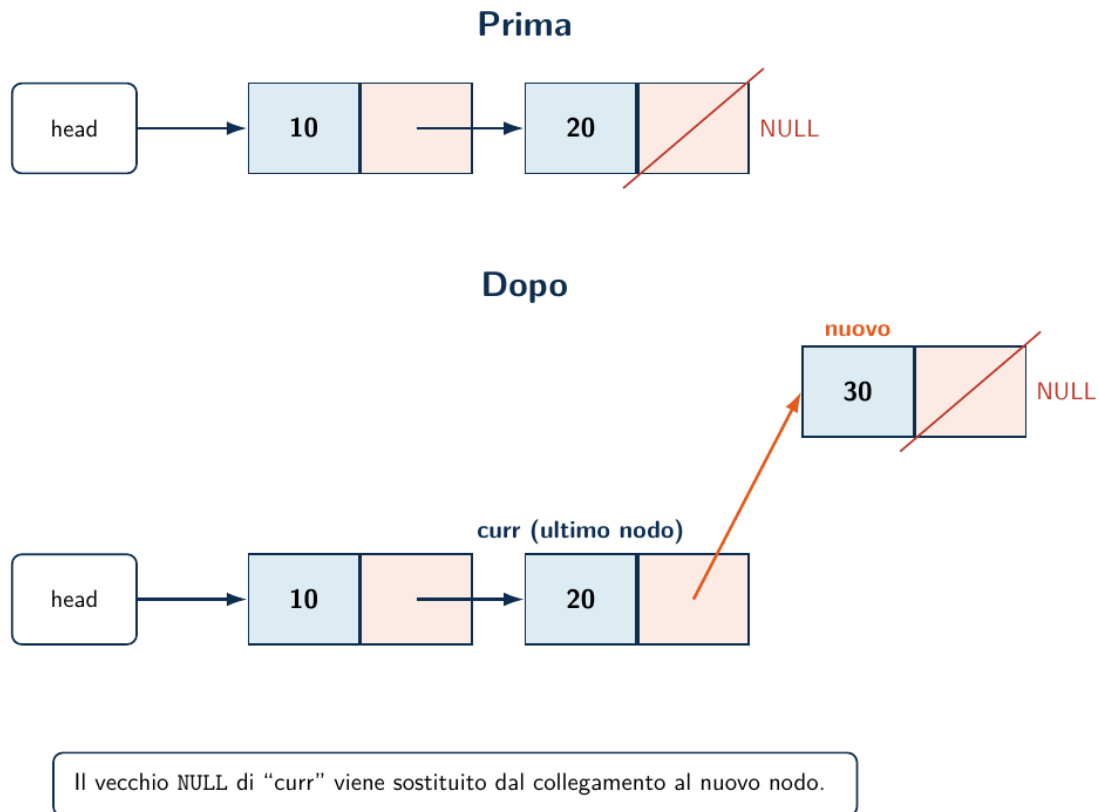


Figura 15.7: Inserimento in coda

```

14  while (corrente != NULL && corrente->dato < valore) {
15      precedente = corrente;
16      corrente = corrente->prossimo;
17  }
18
19  // Chirurgia di collegamento
20  precedente->prossimo = nuovo;
21  nuovo->prossimo = corrente;
22
23  return testa;
24  }

```

15.4 La Chirurgia dei Puntatori: Cancellazioni

La rimozione di un nodo richiede di bypassarlo (scavalcarlo) modificando il puntatore del nodo precedente, ed eliminando fisicamente l'elemento dallo Heap tramite free.

Attenzione

Pericolo di Memory Leak È tassativo salvare l'indirizzo del nodo da eliminare in un puntatore di supporto *prima* di modificare le connessioni. Se si effettua la free del nodo senza preservare il suo campo *prossimo*, si perde il collegamento con tutti i

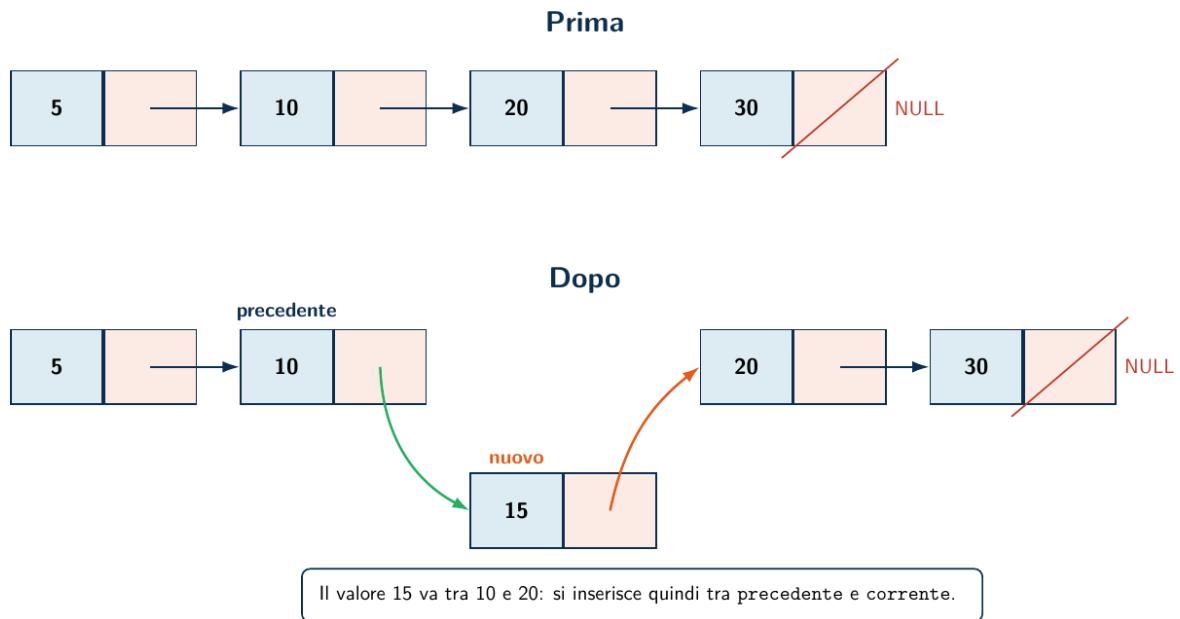


Figura 15.8: Inserimento ordinato

nodi successivi, rendendoli orfani e inaccessibili nello Heap.

15.4.1 Cancellazione in Testa

```

1  Nodo* rimuovi_testa(Nodo *testa) {
2      if (testa == NULL) return NULL;
3
4      Nodo *da_liberare = testa; // Salvataggio riferimento
5      testa = testa->prossimo;   // Spostamento testa
6      free(da_liberare);        // Rimozione sicura
7      return testa;
8  }

```

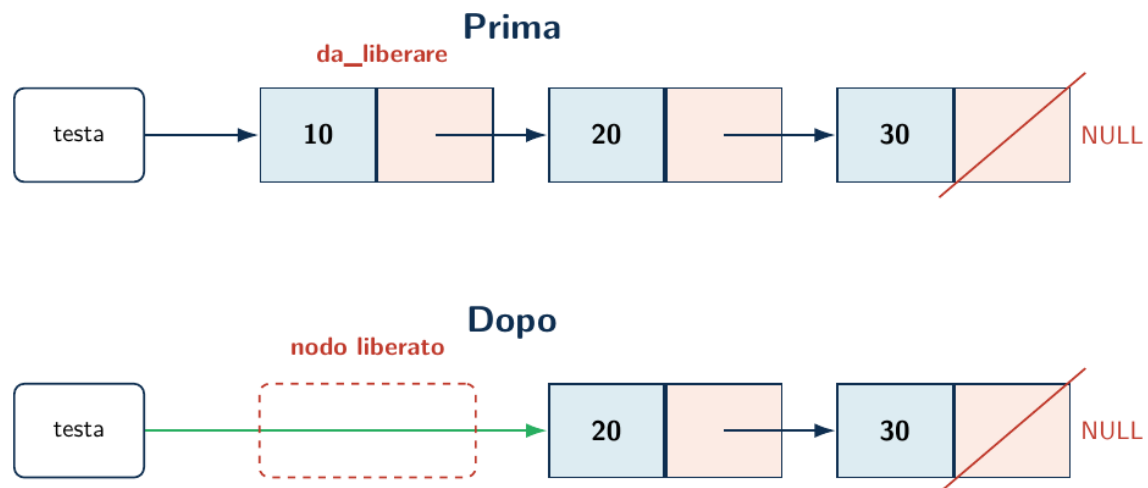
15.4.2 Cancellazione in Coda

Per rimuovere l'ultimo elemento della lista, occorre scorrere l'intera lista per posizionarsi sul penultimo nodo. Una volta individuato il penultimo nodo, si imposta il suo campo `prossimo` a `NULL` e si libera la memoria dell'ultimo nodo.

```

1  Nodo* rimuovi_coda(Nodo *testa) {
2      if (testa == NULL) return NULL;
3
4      // Se c'è solo un nodo
5      if (testa->prossimo == NULL) {
6          free(testa);
7          return NULL;
8      }

```



testa ora punta direttamente al secondo nodo: il primo nodo è stato deallocato.

Figura 15.9: Cancellazione in testa

```

9
10     Nodo *precedente = NULL;
11     Nodo *corrente = testa;
12     while (corrente->prossimo != NULL) {
13         precedente = corrente;
14         corrente = corrente->prossimo;
15     }
16     precedente->prossimo = NULL;
17     free(corrente);
18     return testa;
19 }

```

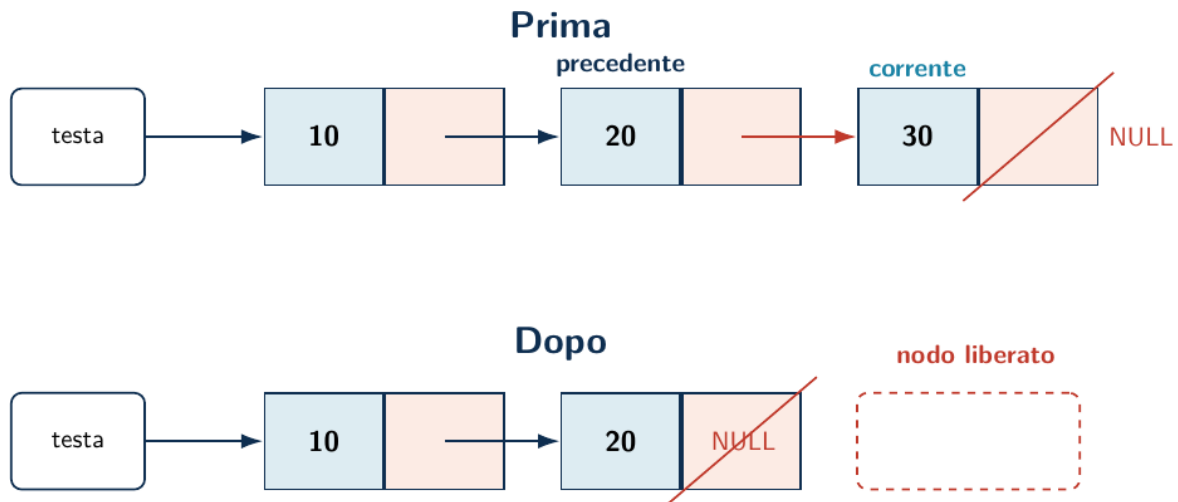
15.4.3 Cancellazione di un Elemento Specifico

Per rimuovere un nodo contenente un dato valore, si esegue una scansione lineare con due puntatori sfalsati. Una volta individuato il valore, si esegue lo scavallo del puntatore (`precedente->prossimo = corrente->prossimo`).

```

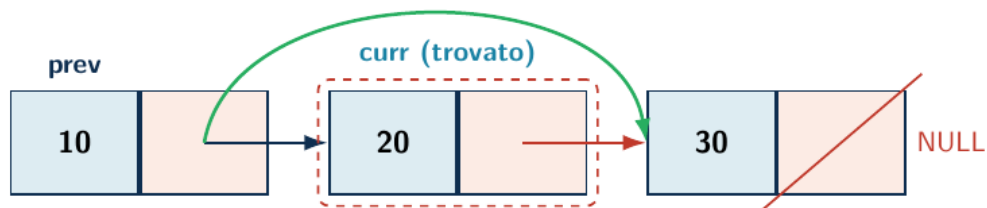
1  Nodo* rimuovi_valore(Nodo *testa, int valore) {
2      if (testa == NULL) return NULL;
3
4      // Rimozione in testa
5      if (testa->dato == valore) {
6          Nodo *nuova_testa = testa->prossimo;
7          free(testa);
8          return nuova_testa;
9      }
10
11     Nodo *precedente = testa;
12     Nodo *corrente = testa->prossimo;

```

Il puntatore prossimo di *precedente* diventa NULL: il vecchio ultimo nodo è stato deallocato.

Figura 15.10: Cancellazione in coda



Il ciclo di ricerca si ferma su `curr->dato == 20`; poi si esegue il detach come nella scheda 9.

Figura 15.11: Ricerca e cancellazione

```

13
14  while (corrente != NULL && corrente->dato != valore) {
15      precedente = corrente;
16      corrente = corrente->prossimo;
17  }
18
19  if (corrente != NULL) {
20      precedente->prossimo = corrente->prossimo; // Detach
21      free(corrente); // Deallocazione
22  }
23
24  return testa;
25  }
```

15.4.4 Deallocazione dell'Intera Lista

Sebbene i moderni sistemi operativi liberino in automatico lo spazio di indirizzamento utilizzato da un programma, di fatto liberando lo spazio delle free non effettuate al momento della chiusura del programma, in ambienti di programmazione di basso livello (es. microcontrollori), non liberare la memoria può dare vita a memory leak persistenti anche alla chiusura del programma. Prima del termine del programma, è pertanto buona norma distruggere ogni lista allocata dinamicamente:

```
1 void svuota_lista(Nodo *testa) {
2     Nodo *corrente = testa;
3     while (corrente != NULL) {
4         Nodo *successore = corrente->prossimo; // Preserva il futuro
5         free(corrente);                       // Dealloca il presente
6         corrente = successore;                 // Avanza
7     }
8 }
```

15.4.5 Ordinamento su Liste (Bubble Sort)

Ora che sappiamo navigare e modificare una lista, come facciamo ad ordinarne una nata disordinata? Negli array usavamo il Bubble Sort scambiando gli elementi con un algoritmo di swap. Sulle liste abbiamo due opzioni:

- **Opzione A (Swap dei Dati):** Copiamo e scambiamo unicamente il valore contenuto nei campi dato dei nodi, lasciando i nodi fisicamente dove si trovano nello Heap.
- **Opzione B (Swap dei Puntatori):** Sganciamo fisicamente i nodi e riallineiamo i loro puntatori prossimo per scambiarne l'ordine.

La Scelta Ingegneristica

Scambiare i puntatori (Opzione B) è un'operazione complessa che richiede di aggiornare fino a 4 indirizzi diversi per ogni singolo swap, aumentando drasticamente il rischio di perdere nodi o creare cicli infiniti. Per garantire la stabilità del codice e la semplicità didattica, l'**Opzione A (Swap dei Dati)** è la scelta ottimale, in quanto ricalca in modo pulito ed intuitivo la logica del Bubble Sort sugli array!

L'algoritmo di Bubble Sort applicato alla Lista

Sostituiamo gli indici numerici `[i]` e `[i+1]` con i puntatori di navigazione `corrente` e `corrente->prossimo`:

```
1 void ordina_lista(Nodo *testa) {
2     // Caso banale: lista vuota o con un solo elemento (gia' ordinata)
3     if (testa == NULL || testa->prossimo == NULL) {
4         return;
5     }
6
7     bool scambiato;
```

```

8     Nodo *corrente;
9     Nodo *limite = NULL; // Ottimizzazione: delimita la fine della
    ↪ scansione
10
11     do {
12         scambiato = false;
13         corrente = testa;
14
15         // Scorriamo la lista fino al limite degli elementi già
    ↪ consolidati
16         while (corrente->prossimo != limite) {
17             // Confrontiamo elementi adiacenti sequenziali
18             if (corrente->dato > corrente->prossimo->dato) {
19                 // LA CHIRURGIA DEL PAYLOAD: Scambio in-place dei dati
20                 int temp = corrente->dato;
21                 corrente->dato = corrente->prossimo->dato;
22                 corrente->prossimo->dato = temp;
23
24                 scambiato = true; // Segnaliamo che è avvenuto uno
    ↪ scambio
25             }
26             corrente = corrente->prossimo; // Passo in avanti
27         }
28         // L'ultimo elemento esaminato è ora sicuramente al suo posto
    ↪ corretto
29         limite = corrente;
30
31     } while (scambiato); // Se non ci sono stati scambi, l'intera lista è
    ↪ ordinata!
32 }

```

15.5 Quiz di Autovalutazione

Quiz 1

Domanda

1. Come si inizializza correttamente una lista vuota in C?

- A) Dichiarando un puntatore a Nodo e assegnandogli il valore NULL (es. `Nodo *testa = NULL;`).
- B) Allocando subito un nodo vuoto tramite `malloc`.
- C) Dichiarando una variabile `Nodo` locale nello Stack senza inizializzarla.
- D) Impostando il valore del primo nodo a 0.

Soluzione
Risposta corretta: A
Spiegazione: Per indicare che una lista non contiene alcun elemento, il puntatore di testa (testa) deve essere impostato esplicitamente a NULL. Questo evita che contenga indirizzi spazzatura e consente alle funzioni di controllo di rilevarne lo stato vuoto.

Quiz 2

Domanda

2. Qual è l'effetto di perdere l'indirizzo memorizzato nella variabile di testa head?

- A) La lista si inverte automaticamente in memoria.
- B) Un memory leak irreversibile di tutti i nodi della lista.
- C) I nodi vengono deallocati automaticamente dal Garbage Collector di C.
- D) Il programma continua a funzionare regolarmente accedendo tramite indici.

Soluzione
Risposta corretta: B
Spiegazione: Nello Heap, i nodi sono accessibili unicamente conoscendo l'indirizzo del primo nodo (head). Se questa variabile viene persa o sovrascritta senza prima aver deallocato la memoria, gli indirizzi dei nodi diventano intracciabili per la CPU, bloccando quella porzione di RAM fino al termine del processo.

Quiz 3

Domanda

3. Cosa contiene il campo prossimo dell'ultimo nodo di una lista ben formata?

- A) L'indirizzo della testa (head).
- B) Un valore casuale sporco di memoria (*garbage value*).
- C) Il valore speciale NULL.
- D) L'indirizzo del nodo stesso.

Soluzione
Risposta corretta: C
Spiegazione: In C, NULL indica l'assenza di un indirizzo di memoria valido ed è utilizzato come sentinella standard per arrestare i cicli di scorrimento delle liste.

Quiz 4

Domanda

4. Quale operazione sui puntatori è necessaria per inserire un nuovo nodo in testa a una lista non vuota?

- A) Far puntare il campo prossimo del nuovo nodo a NULL e deallocare la testa.
- B) Far puntare il campo prossimo del nuovo nodo al nodo che era precedentemente in testa, e poi aggiornare il puntatore di testa per farlo puntare al nuovo nodo.
- C) Spostare tutti i valori della lista nei nodi adiacenti verso destra.
- D) Modificare l'indirizzo del puntatore di testa lasciando inalterato il campo prossimo del nuovo nodo.

Soluzione
Risposta corretta: B
Spiegazione: Per inserire un nodo in testa senza perdere il collegamento con i nodi già esistenti, il nuovo nodo deve prima "agganciarsi" alla vecchia testa salvandone l'indirizzo nel suo campo prossimo. Solo a questo punto possiamo muovere il puntatore di testa sul nuovo nodo.

Quiz 5

Domanda

5. Perché nell'allocazione dinamica di un nodo è importante verificare il valore di ritorno della `malloc`?

- A) Per evitare memory leaks.
- B) Perché la `malloc` restituisce NULL in caso di memoria Heap esaurita, provocando crash (Segmentation Fault) in caso di mancato controllo.
- C) Per verificare che il dato inserito sia di tipo numerico.
- D) Per liberare automaticamente il nodo deallocato in precedenza.

Soluzione
Risposta corretta: B
Spiegazione: Tentare di dereferenziare un puntatore NULL (es. `nuovo->dato = valore` quando nuovo è NULL) provoca un crash immediato del programma (Segmentation Fault).

Quiz 6

Domanda

6. Nella funzione `inserisci\{}_coda`, perché il ciclo di scorrimento si ferma quando

corrente->prossimo != NULL anziché corrente != NULL?

- A) Perché altrimenti si verificherebbe un ciclo infinito.
- B) Per evitare di allocare due nodi contemporaneamente.
- C) Perché abbiamo bisogno di fermarci esattamente sull'ultimo nodo per poterne modificare il campo `prossimo`.
- D) Perché l'ultimo nodo ha il puntatore al dato pari a 0.

Soluzione
Risposta corretta: C
Spiegazione: Se ci fermassimo quando `corrente == NULL`, avremmo superato la lista e non avremmo più un nodo valido su cui innestare il puntatore al nuovo elemento.

Quiz 7

Domanda

7. Nella rimozione di un nodo intermedio, perché sono necessari due puntatori (precedente e corrente)?

- A) Per raddoppiare la velocità di esecuzione della cancellazione.
- B) Perché la lista semplice è monodirezionale e serve `precedente` per scavalcare `corrente` una volta trovato il valore.
- C) Per gestire contemporaneamente l'inserimento e la rimozione.
- D) Perché non è possibile usare variabili locali all'interno della funzione.

Soluzione
Risposta corretta: B
Spiegazione: Una volta posizionati sul nodo da cancellare (`corrente`), non abbiamo alcun modo di risalire al nodo che lo precede per fare in modo che quest'ultimo "scavalchi" l'elemento rimosso.

Quiz 8

Domanda

8. Che cosa accade se si invoca `free(testa)` come prima istruzione di una funzione di deallocazione totale della lista?

- A) La memoria viene liberata correttamente e `head` viene aggiornato.
- B) Si perde l'indirizzo del nodo successivo, impedendo di scorrere e deallocare il resto della lista (causando leak).
- C) Il compilatore segnala un errore di sintassi bloccante.

- D) Tutti i nodi successivi vengono automaticamente spostati nello Stack.

Soluzione
Risposta corretta: B
Spiegazione: Liberare il nodo distrugge il campo `prossimo`. Bisogna sempre salvare l'indirizzo del successore in una variabile temporanea prima di deallocare il nodo corrente.

Quiz 9

Domanda

9. Quale delle seguenti dichiarazioni definisce correttamente un tipo nodo autoreferenziale?

- A) `typedef struct \{\} int dato; Nodo *prossimo; \{\} Nodo;`
- B) `typedef struct Nodo \{\} int dato; struct Nodo *prossimo; \{\} Nodo;`
- C) `typedef struct Nodo \{\} int dato; Nodo prossimo; \{\} Nodo;`
- D) `struct Nodo \{\} int dato; struct Nodo prossimo; \{\};`

Soluzione
Risposta corretta: B
Spiegazione: Il compilatore ha bisogno di conoscere l'esistenza di `struct Nodo` per dichiarare il puntatore interno prima che la definizione del tipo alias `Nodo` sia completata.

Quiz 10

Domanda

10. Qual è il principale svantaggio delle liste concatenate rispetto agli array statici?

- A) Il fatto che non possono memorizzare valori negativi.
- B) Mancanza dell'accesso casuale e consumo extra di memoria per memorizzare i puntatori.
- C) L'impossibilità di usare tipi strutturati (es. `struct`) come payload del nodo.
- D) Il rischio costante di esaurire lo Stack di sistema a causa dell'allocazione dinamica.

Soluzione
Risposta corretta: B
Spiegazione: Per ogni elemento salvato in lista, occorre consumare dello spazio aggiuntivo in memoria (4 o 8 byte) per salvare l'indirizzo del nodo successivo.

15.6 Esercizi Pratici Risolti

Esercizio 1: Conteggio dei Nodi

Domanda

Scrivere una funzione `int conta_nodi(Nodo *testa)` che restituisca il numero totale di elementi presenti nella lista.

```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  int conta_nodi(Nodo *testa) {
9      int conteggio = 0;
10     Nodo *corrente = testa;
11     while (corrente != NULL) {
12         conteggio++;
13         corrente = corrente->prossimo;
14     }
15     return conteggio;
16 }
```

Soluzione

Esercizio 2: Somma e Media dei Nodi

Domanda

Scrivere una funzione `double media_lista(Nodo *testa)` che calcoli la media aritmetica dei valori interi contenuti nei nodi di una lista semplice. Nel caso di lista vuota, la funzione deve restituire `0.0`.


```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  double media_lista(Nodo *testa) {
9      if (testa == NULL) {
10         return 0.0;
11     }
12     int somma = 0;
13     int conteggio = 0;
14     Nodo *corrente = testa;
15     while (corrente != NULL) {
16         somma += corrente->dato;
17         conteggio++;
18         corrente = corrente->prossimo;
19     }
20     return (double)somma / conteggio;
21 }

```

Soluzione**Esercizio 3: Verifica Presenza Elemento (Ricerca Lineare)****Domanda**

Scrivere una funzione `bool contiene({}_valore(Nodo *testa, int target)` che restituisca `true` se il valore `target` è presente in almeno uno dei nodi della lista, e `false` altrimenti.

```

1  #include <stdbool.h>
2  #include <stdio.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  bool contiene_valore(Nodo *testa, int target) {
10     Nodo *corrente = testa;
11     while (corrente != NULL) {
12         if (corrente->dato == target) {
13             return true; // Trovato, interruzione anticipata
14         }
15         corrente = corrente->prossimo;
16     }
17     return false; // Elemento non presente
18 }

```

Soluzione**Esercizio 4: Inserimento in Testa In-Place (Doppio****Domanda**

Puntatore)

Implementare la funzione `void inserisci\{\}_testa\{\}_doppio\{\}_ptr(Nodo **testa\{\}_ptr, int valore)` che inserisce un elemento in testa alla lista modificando direttamente la variabile di testa del chiamante passata tramite riferimento.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  void inserisci_testa_doppio_ptr(Nodo **testa_ptr, int valore) {
10     Nodo *nuovo = (Nodo*) malloc(sizeof(Nodo));
11     if (nuovo == NULL) {
12         exit(EXIT_FAILURE);
13     }
14     nuovo->dato = valore;
15
16     // Il nuovo nodo punta al nodo attualmente referenziato dalla
17     // testa
18     nuovo->prossimo = *testa_ptr;
19
20     // Aggiorniamo il valore del puntatore originale nello Stack
21     *testa_ptr = nuovo;
22 }

```

Soluzione**Esercizio 5: Deallocazione Totale di una Lista****Domanda**

Scrivere una funzione `void distruggi_list(Nodo **testa_ptr)` che liberi tutta la memoria occupata dai nodi della lista e, al termine, imposti il puntatore originale del chiamante a NULL per evitare puntatori pendenti (*dangling pointers*).

Analisi algoritmica: Si liberano individualmente gli N nodi allocati dinamicamente. Complessità temporale: $O(N)$. Azzerando `*testa\{\}_ptr` garantiamo la sicurezza del codice chiamante che non tenterà di accedere a memoria non più valida.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  void distruggi_lista(Nodo **testa_ptr) {
10     if (*testa_ptr == NULL) return;
11     Nodo *corrente = *testa_ptr;
12     while (corrente != NULL) {
13         Nodo *prossimo = corrente->prossimo; // Conserviamo il link
14         // Deallocazione sicura
15         free(corrente); // del presente
16         corrente = prossimo; // Avanzamento
17     }
18     *testa_ptr = NULL; // Azzerare la variabile originale del
19                             ↪ chiamante
20 }

```

Soluzione

Esercizio 6: Inversione In-Place di una Lista (Reverse)

Domanda

Scrivere una funzione `Nodo* inverti\{\}_lista(Nodo *testa)` che inverta l'ordine dei collegamenti dei nodi in modo che l'ultimo nodo diventi la nuova testa, e il primo nodo punti a NULL. L'operazione deve essere eseguita *in-place*, ovvero senza allocare nuovi nodi.

```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  Nodo* inverti_lista(Nodo *testa) {
9      Nodo *precedente = NULL;
10     Nodo *corrente = testa;
11     Nodo *successore = NULL;
12
13     while (corrente != NULL) {
14         successore = corrente->prossimo; // Salva il resto della
15         ↪ lista
16         corrente->prossimo = precedente; // Inverte la direzione
17         ↪ del puntatore
18         precedente = corrente;
19         corrente = successore;
20     }
21     return precedente; // Nuova testa della lista invertita

```

Soluzione**Esercizio 7: Riconoscimento ed Eliminazione dei Duplicati****Domanda**

in Lista Ordinata

Scrivere una funzione `void rimuovi\_duplicati\_ordinata(Nodo *testa)` che, data una lista ordinata in modo crescente, rimuova tutti i nodi duplicati adiacenti mantenendo una sola copia di ciascun valore.

Analisi algoritmica: Poiché la lista è già ordinata, i duplicati sono necessariamente vicini. La scansione richiede tempo $O(N)$ ed effettua la free dei soli elementi ridondanti.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  void rimuovi_duplicati_ordinata(Nodo *testa) {
10     if (testa == NULL) return;
11     Nodo *corrente = testa;
12     while (corrente->prossimo != NULL) {
13         if (corrente->dato == corrente->prossimo->dato) {
14             // Rilevato duplicato adiacente
15             Nodo *doppione = corrente->prossimo;
16             corrente->prossimo = doppione->prossimo; // Scavalamento
17             free(doppione);
18         } else {
19             corrente = corrente->prossimo; // Avanziamo solo se non
20             // c'era duplicato
21         }
22     }
23 }

```

Soluzione

Esercizio 8: Unione di Due Liste Ordinate (Merge)

Domanda

Scrivere una funzione `Nodo* fondi({}_liste){}_ordinate(Nodo *l1, Nodo *l2)` che prenda in input due liste ordinate in modo crescente e restituisca una terza lista ordinata contenente l'unione dei nodi delle due liste originali. È possibile staccare e riutilizzare i nodi esistenti.

```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  Nodo* fondi_liste_ordinate(Nodo *l1, Nodo *l2) {
9      // Casi base se una delle due liste è vuota
10     if (l1 == NULL) return l2;
11     if (l2 == NULL) return l1;
12     Nodo *nuova_testa = NULL;
13
14     // Determina il primo nodo della nuova lista
15     if (l1->dato <= l2->dato) {
16         nuova_testa = l1;
17         l1 = l1->prossimo;
18     } else {
19         nuova_testa = l2;
20         l2 = l2->prossimo;
21     }
22
23     Nodo *corrente = nuova_testa;
24
25     // FONDE confrontando gli elementi testa a testa
26     while (l1 != NULL && l2 != NULL) {
27         if (l1->dato <= l2->dato) {
28             corrente->prossimo = l1;
29             l1 = l1->prossimo;
30         } else {
31             corrente->prossimo = l2;
32             l2 = l2->prossimo;
33         }
34     }
35     corrente = corrente->prossimo;
36
37     // Collega l'eventuale coda rimanente di una delle due liste
38     if (l1 != NULL) {
39         corrente->prossimo = l1;
40     } else {
41         corrente->prossimo = l2;
42     }
43
44     return nuova_testa;
45 }
46

```

Soluzione

Esercizio 9: Individuazione del Nodo Mediano (Puntatori)

Domanda

Veloce e Lento)

Implementare una funzione `Nodo* trova_{}_mediano(Nodo *testa)` che restituisca l'indirizzo di memoria del nodo centrale di una lista semplice senza calcolarne preventivamente la lunghezza totale. Se la lista ha lunghezza pari, si consideri il primo dei due nodi centrali.

```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  Nodo* trova_mediano(Nodo *testa) {
9      if (testa == NULL) return NULL;
10     Nodo *lento = testa;
11     Nodo *veloce = testa;
12
13     // Il puntatore veloce cammina al doppio della velocità del
14     // lento
15     while (veloce->prossimo != NULL && veloce->prossimo->prossimo
16            != NULL) {
17         lento = lento->prossimo;
18         veloce = veloce->prossimo->prossimo;
19     }
20     return lento; // Quando veloce arriva alla fine, lento è a metà?
21 }
```

Soluzione

Esercizio 10: Concatenazione in Coda di Due Liste (Append)

Domanda

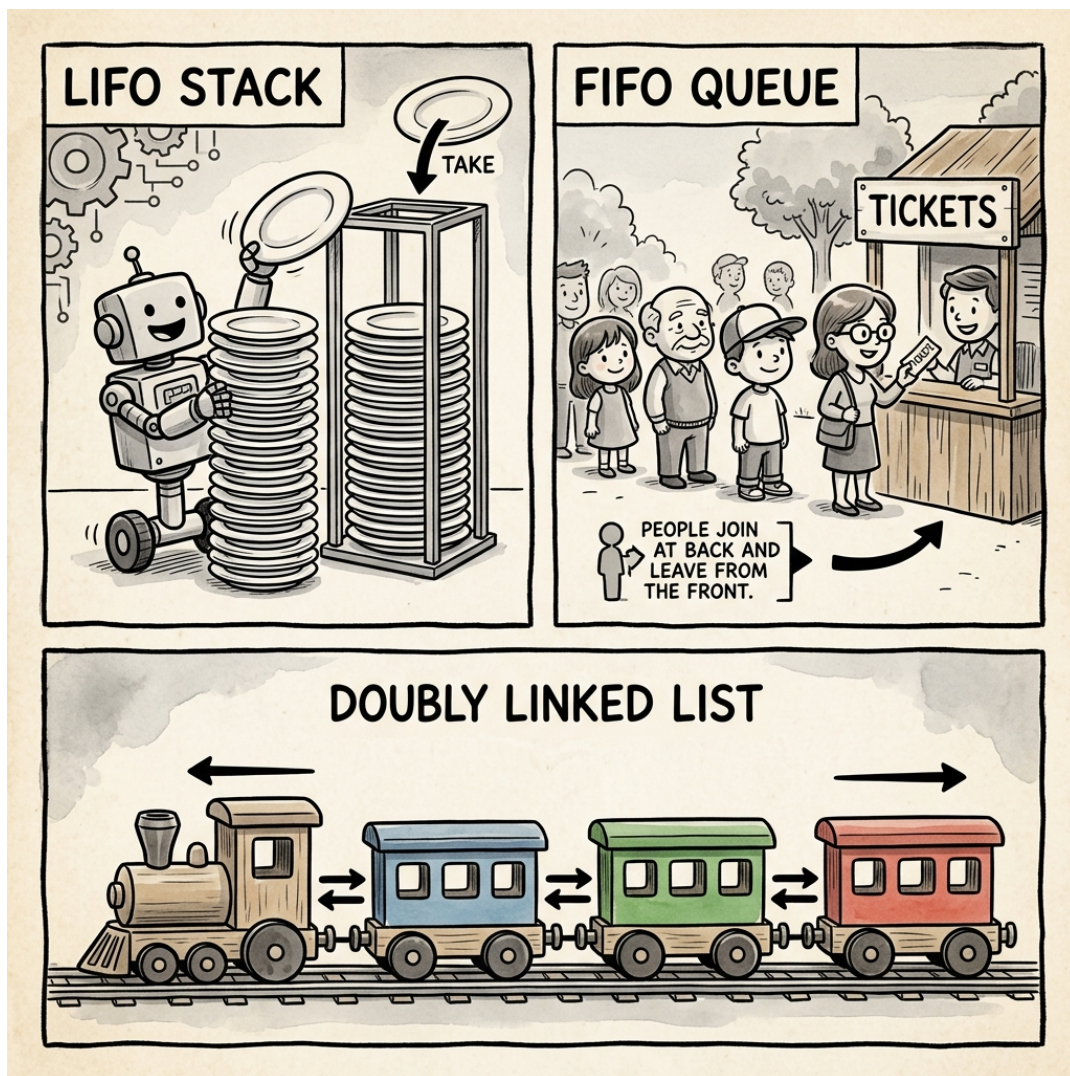
Scrivere una funzione `Nodo* concatena_{}_liste(Nodo *l1, Nodo *l2)` che colleghi la lista `l2` in coda alla lista `l1`. La funzione deve restituire la testa della nuova lista unificata.

Soluzione

```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  Nodo* concatena_liste(Nodo *l1, Nodo *l2) {
9      // Caso speciale: se la prima lista è vuota, restituiamo
10     ↪ direttamente la seconda
11     if (l1 == NULL) {
12         return l2;
13     }
14     Nodo *corrente = l1;
15     // Raggiungiamo l'ultimo nodo di l1
16     while (corrente->prossimo != NULL) {
17         corrente = corrente->prossimo;
18     }
19     // Aggancio del primo nodo di l2 alla coda di l1
20     corrente->prossimo = l2;
21     return l1;
22 }
23
24 }
```

Pile, Code e Liste Doppie



Lezione 16

Pile, Code e Liste Doppie

Nelle lezioni precedenti abbiamo visto come memorizzare dati all'interno di strutture lineari come gli array (allocazione contigua in RAM) e le liste concatenate semplici (nodi sparsi nello Heap collegati in avanti). In questa lezione facciamo un passo avanti: separiamo la rappresentazione fisica dei dati dalle regole logiche con cui vi accediamo, introducendo il concetto di **Tipo di Dato Astratto (ADT)**. Studieremo poi come implementare in C due delle strutture dati più importanti dell'informatica, la **Pila (Stack)** e la **Coda (Queue)**, concludendo con un'introduzione alle **Liste Doppiaemente Concatenate**.

16.1 Il Concetto di Tipo di Dato Astratto (ADT)

Fino ad ora abbiamo manipolato i dati preoccupandoci di come disporli fisicamente in memoria. Ad esempio, per rappresentare una sequenza di interi, abbiamo usato un array (contiguo) o una lista (frammentata).

Un **Tipo di Dato Astratto** (*Abstract Data Type - ADT*) è una descrizione logica di una struttura dati che definisce:

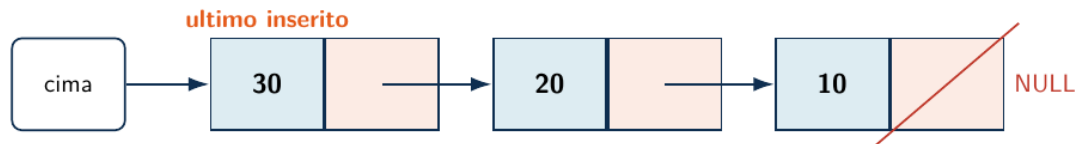
1. I dati memorizzati.
2. Le operazioni consentite su di essi.
3. Le regole con cui è permesso accedere a tali dati.

In un ADT, l'implementazione fisica (array o lista) viene nascosta all'utente esterno dietro un'**interfaccia pubblica** (un insieme di funzioni). Questo principio di programmazione prende il nome di **incapsulamento** ed **astrazione dei dati**.

16.2 La Pila (Stack)

La **Pila** (o *Stack*) è una struttura dati lineare governata dalla logica **LIFO** (*Last-In, First-Out*): l'ultimo elemento inserito è sempre il primo ad essere rimosso.

Pila con tre elementi (inseriti in ordine 10, 20, 30)



L'elemento 30, inserito per ultimo, è il primo raggiungibile da cima: sarà anche il primo ad essere rimosso.

Figura 16.1: Struttura della Pila

16.2.1 Metafore e Casi d'Uso

- **Metafora reale:** Una pila di piatti in un dispenser a molla: possiamo appoggiare un piatto solo sopra la cima, e possiamo prelevare solo l'ultimo piatto posato in cima.
- **Applicazioni informatiche:**
 - Lo Stack dei Record di Attivazione delle funzioni gestito a livello hardware della CPU.
 - La funzionalità “Annulla” (*Undo* o *Ctrl+Z*) dei programmi di videoscrittura.
 - La cronologia di navigazione dei browser web (il tasto “Indietro” torna all’ultima pagina visitata).

16.2.2 Implementazione di una Pila in C

Per implementare una pila dinamica senza sprechi di memoria, utilizziamo una lista concatenata semplice. Facciamo coincidere la “cima” della pila con la **testa** della lista. In questo modo, sia l’inserimento (*push*) che la rimozione (*pop*) avvengono in testa alla lista in tempo costante $O(1)$, evitando di scorrere i nodi.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4
5  // Definizione del nodo
6  typedef struct Nodo {
7      int dato;
8      struct Nodo *prossimo;
9  } Nodo;
10
11 // Definizione del tipo astratto Pila
12 typedef Nodo* Pila;
13

```

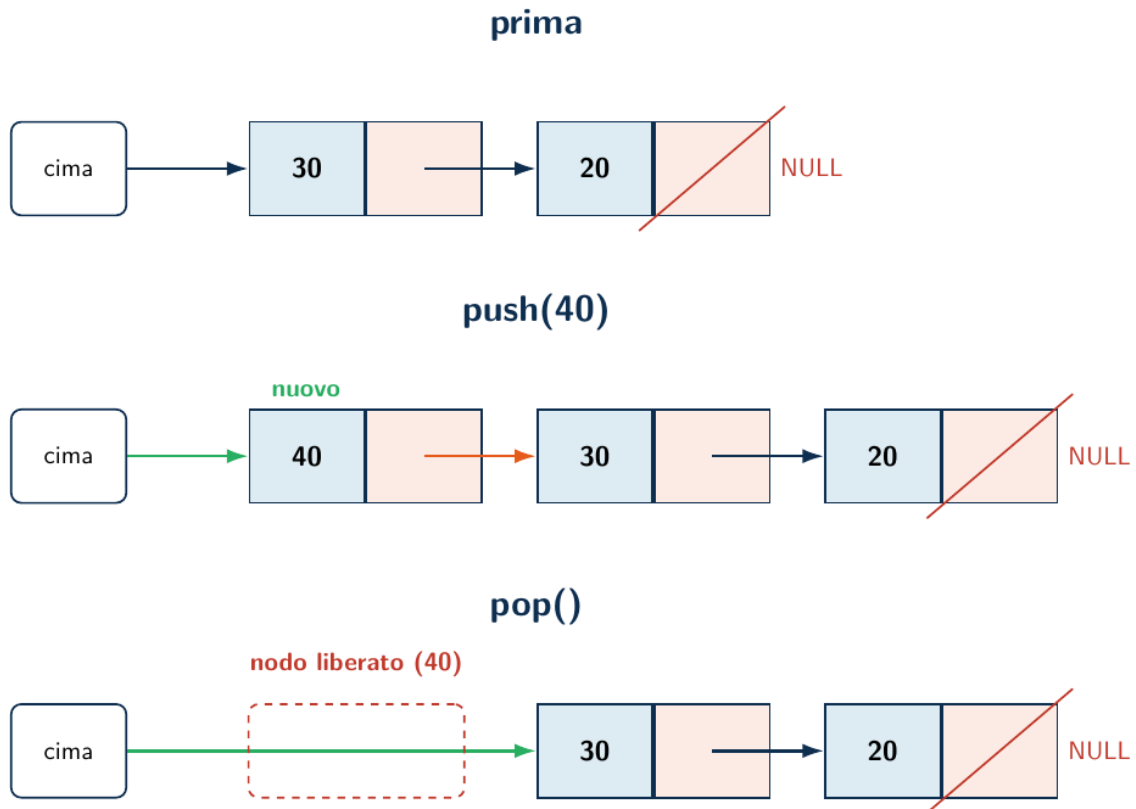


Figura 16.2: Operazioni di Push e Pop

```

14 // Inizializza una pila vuota
15 Pila crea_pila(void) {
16     return NULL;
17 }
18
19 // Verifica se la pila ? vuota
20 bool is_empty_pila(Pila p) {
21     return p == NULL;
22 }

```

Operazioni Push, Pop e Peek

Le operazioni di push e pop permettono di inserire ed eliminare elementi dalla pila, come mostrato di seguito:

L'operazione peek permette di leggere l'elemento in cima alla pila senza eliminarlo dalla struttura.

Operazione Push Inserisce un elemento in cima alla pila (equivalente a un inserimento in testa alla lista).

```

1 Pila push(Pila p, int valore) {
2     Nodo *nuovo = (Nodo*) malloc(sizeof(Nodo));
3     if (nuovo == NULL) {

```

```
4         fprintf(stderr, "Errore: Memoria esaurita\n");
5         exit(1);
6     }
7     nuovo->dato = valore;
8     nuovo->prossimo = p; // Il nuovo nodo punta alla vecchia cima
9     return nuovo;       // Il nuovo nodo diventa la nuova cima della
    ↪ pila
10 }
```

Operazione Pop Estrae l'elemento in cima alla pila, ne restituisce il valore attraverso un puntatore e dealloca il nodo rimosso.

```
1 bool pop(Pila *p, int *valore_estratto) {
2     if (is_empty_pila(*p)) {
3         return false; // Underflow della pila (pila vuota)
4     }
5
6     Nodo *da_eliminare = *p;
7     *valore_estratto = da_eliminare->dato;
8
9     *p = (*p)->prossimo; // Sposta la cima sul nodo successivo
10    free(da_eliminare); // Dealloca la memoria
11    return true;
12 }
```

Operazione Peek Consente di ispezionare il valore in cima alla pila senza rimuoverlo.

```
1 bool peek(Pila p, int *valore_cima) {
2     if (is_empty_pila(p)) {
3         return false;
4     }
5     *valore_cima = p->dato;
6     return true;
7 }
```

16.3 La Coda (Queue)

La **Coda** (o *Queue*) è una struttura dati lineare governata dalla logica **FIFO** (*First-In, First-Out*): il primo elemento inserito è il primo ad essere rimosso.

16.3.1 Metafore e Casi d'Uso

- **Metafora reale:** La fila ordinata alle casse di un supermercato: chi arriva per primo viene servito per primo, chi si accoda per ultimo deve attendere il proprio turno.
- **Applicazioni informatiche:**
 - La coda di stampa del sistema operativo.

Coda con tre elementi (inseriti in ordine 10, 20, 30)

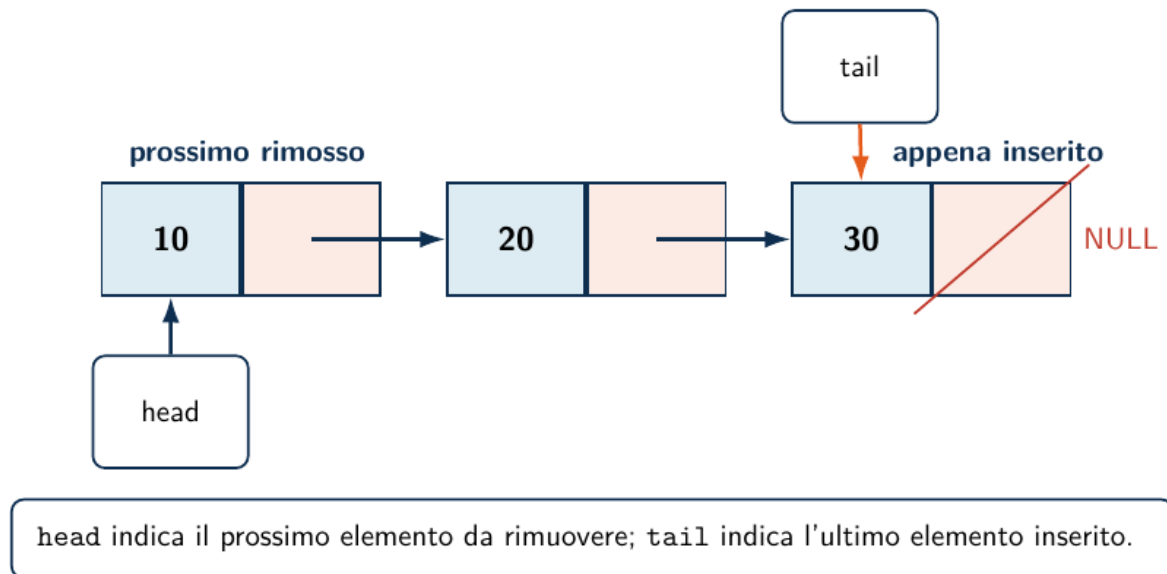


Figura 16.3: Struttura della Coda

- La trasmissione dei pacchetti di dati nei buffer di memoria dei router di rete.

16.3.2 Testa, Coda ed Efficienza

Se implementiamo una coda usando una normale lista semplice memorizzando solo il puntatore di testa:

- Se estraiamo (dequeue) in testa alla lista, l'operazione richiede tempo costante $O(1)$.
- Se inseriamo (enqueue) in coda alla lista, siamo costretti a scorrere l'intera lista fino all'ultimo elemento ad ogni inserimento, con complessità temporale lineare $O(N)$.

Per ovviare a questo problema ed eseguire entrambe le operazioni in tempo costante $O(1)$, ingegnerizziamo una struttura di controllo che mantiene **due puntatori**:

1. Un puntatore head alla testa della lista (per prelevare).
2. Un puntatore tail all'ultimo nodo della lista (per inserire).

16.3.3 Implementazione di una Coda in C

```

1 typedef struct NodoCoda {
2     int dato;
3     struct NodoCoda *prossimo;
4 } NodoCoda;
5 
```

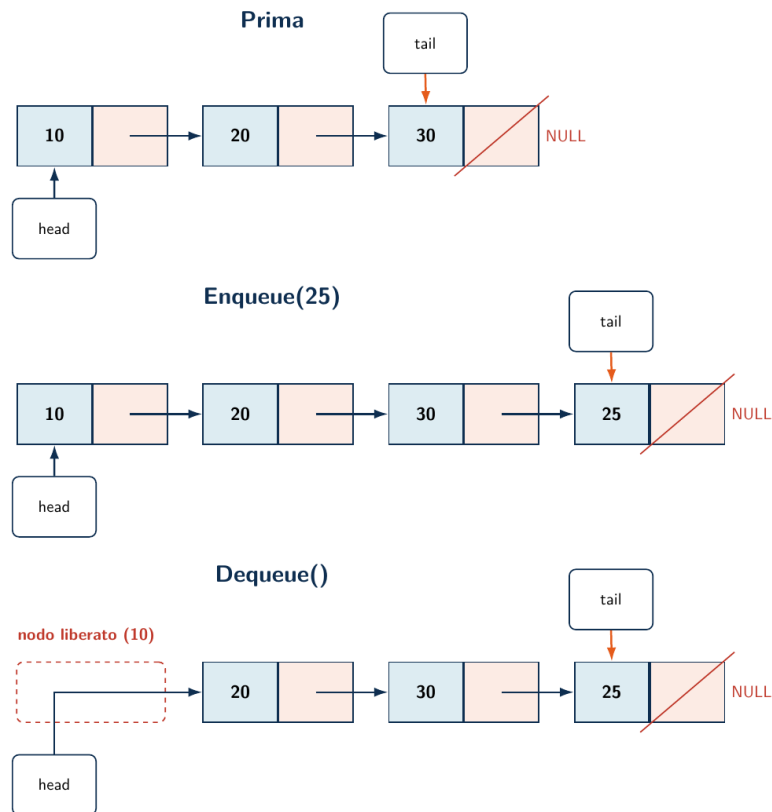


Figura 16.4: Operazioni di Enqueue e Dequeue

```

6 // Struttura di controllo con doppio puntatore
7 typedef struct {
8     NodoCoda *head; // Accesso all'inizio della coda (per dequeue)
9     NodoCoda *tail; // Accesso alla fine della coda (per enqueue)
10 } Coda;
11
12 // Inizializza la coda
13 void inizializza_coda(Coda *c) {
14     c->head = NULL;
15     c->tail = NULL;
16 }
17
18 // Verifica se la coda ? vuota
19 bool is_empty_coda(const Coda *c) {
20     return c->head == NULL;
21 }

```

Operazioni di Enqueue e Dequeue

Le operazioni di enqueue e dequeue permettono di inserire e rimuovere elementi dalla coda, come mostrato di seguito:

Operazione Enqueue Aggiunge un elemento alla fine della coda in tempo $O(1)$ sfruttando il puntatore `tail`.


```

1 void enqueue(Coda *c, int valore) {
2     NodoCoda *nuovo = (NodoCoda*) malloc(sizeof(NodoCoda));
3     if (nuovo == NULL) {
4         fprintf(stderr, "Errore: Memoria esaurita\n");
5         exit(1);
6     }
7     nuovo->dato = valore;
8     nuovo->prossimo = NULL;
9
10    if (is_empty_coda(c)) {
11        // Se la coda ? vuota, il nuovo nodo ? sia testa che coda
12        c->head = nuovo;
13        c->tail = nuovo;
14    } else {
15        // Il vecchio ultimo nodo viene collegato al nuovo
16        c->tail->prossimo = nuovo;
17        // Il puntatore di coda si sposta sul nuovo nodo
18        c->tail = nuovo;
19    }
20 }

```

Operazione Dequeue Estrae l'elemento in testa alla coda in tempo $O(1)$ sfruttando il puntatore head.

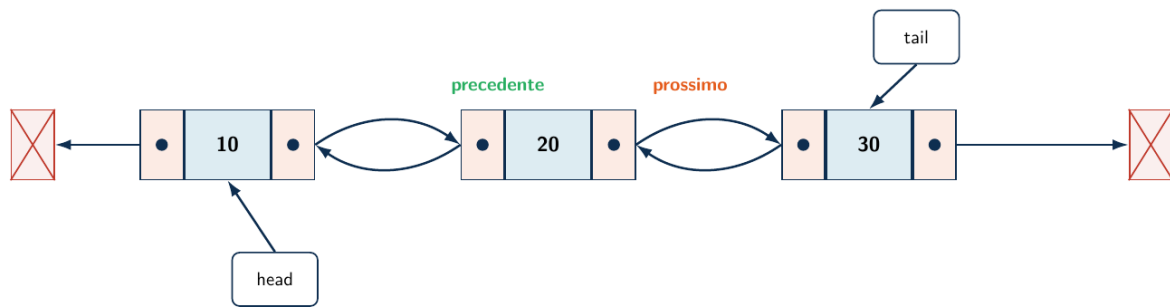
```

1 bool dequeue(Coda *c, int *valore_estratto) {
2     if (is_empty_coda(c)) {
3         return false; // Underflow della coda
4     }
5
6     NodoCoda *da_eliminare = c->head;
7     *valore_estratto = da_eliminare->dato;
8
9     c->head = c->head->prossimo; // La testa avanza al nodo successivo
10
11    // Se la coda si ? svuotata, azzeriamo anche il puntatore tail
12    if (c->head == NULL) {
13        c->tail = NULL;
14    }
15
16    free(da_eliminare); // Deallocazione memoria
17    return true;
18 }

```

16.4 Liste Doppiamente Concatenate

Nelle liste semplici, ogni nodo contiene solo un puntatore in avanti (*prossimo*). Questo rende impossibile retrocedere o effettuare scansioni all'indietro.



head punta sempre al primo nodo, tail punta sempre all'ultimo: insieme permettono accesso immediato a entrambe le estremità della lista.

Figura 16.5: Lista Doppiaemente Concatenata

16.4.1 La Struttura dei Nodi Doppi

Per navigare in entrambe le direzioni, ogni nodo ospita **due puntatori**:

- Un puntatore prossimo che punta in avanti.
- Un puntatore precedente che punta all'indietro.

```

1 typedef struct NodoDoppio {
2     int dato;
3     struct NodoDoppio *precedente; // Collegamento all'indietro
4     struct NodoDoppio *prossimo;   // Collegamento in avanti
5 } NodoDoppio;
6
7 // Struttura di controllo per la gestione ottimizzata
8 typedef struct {
9     NodoDoppio *head; // Inizio lista
10    NodoDoppio *tail; // Fine lista
11 } ListaDoppia;
```

```

1 // Inizializzazione della lista doppia
2 void inizializza_lista_doppia(ListaDoppia *l) {
3     l->head = NULL;
4     l->tail = NULL;
5 }
6
7 // Allocazione del singolo nodo doppio
8 NodoDoppio* crea_nodo_doppio(int valore) {
9     NodoDoppio *nuovo = (NodoDoppio*) malloc(sizeof(NodoDoppio));
10    if (nuovo == NULL) {
11        fprintf(stderr, "Errore: RAM esaurita!\n");
12        exit(1);
13    }
14    nuovo->dato = valore;
15    nuovo->precedente = NULL;
16    nuovo->prossimo = NULL;
17    return nuovo;
18 }
```

16.4.2 Inserimenti Bilaterali

Grazie alla presenza del puntatore `tail` e del collegamento precedente, possiamo inserire elementi sia in testa che in coda alla lista in tempo costante $O(1)$ senza mai dover effettuare scansioni lineari dei nodi.

Inserimento in Testa

```

1 void inserisci_testa_doppia(ListaDoppia *l, int valore) {
2     NodoDoppio *nuovo = crea_nodo_doppio(valore);
3
4     if (l->head == NULL) {
5         // Caso lista vuota: il nuovo nodo ? sia testa che coda
6         l->head = nuovo;
7         l->tail = nuovo;
8     } else {
9         nuovo->prossimo = l->head;    // Il nuovo punta in avanti alla
        ↪ vecchia testa
10        l->head->precedente = nuovo;  // La vecchia testa punta all'
        ↪ indietro al nuovo
11        l->head = nuovo;              // Aggiorniamo la testa di
        ↪ controllo
12    }
13 }
```

Inserimento in Coda

```

1 void inserisci_coda_doppia(ListaDoppia *l, int valore) {
2     NodoDoppio *nuovo = crea_nodo_doppio(valore);
3
4     if (l->head == NULL) {
5         // Caso lista vuota
6         l->head = nuovo;
7         l->tail = nuovo;
8     } else {
9         nuovo->precedente = l->tail; // Il nuovo si allaccia all'
        ↪ indietro al vecchio ultimo
10        l->tail->prossimo = nuovo;    // Il vecchio ultimo punta in
        ↪ avanti al nuovo arrivato
11        l->tail = nuovo;              // Aggiorniamo la coda di
        ↪ controllo
12    }
13 }
```

16.4.3 Rimozione Istantanea

Il grande vantaggio delle liste doppie risiede nell'eliminazione di un nodo. Se si possiede già il puntatore esatto al nodo da eliminare (da `{ }_eliminare`), non occorre alcuna scansione della lista per trovare il predecessore. L'operazione avviene in tempo costante $O(1)$ ricollegando direttamente i puntatori dei nodi adiacenti.

```
1 void rimuovi_nodo_doppio(ListaDoppia *l, NodoDoppio *da_eliminare) {
2     if (l->head == NULL || da_eliminare == NULL) return;
3
4     // Se il nodo da eliminare ? la testa
5     if (l->head == da_eliminare) {
6         l->head = da_eliminare->prossimo;
7     }
8
9     // Se il nodo da eliminare ? la coda
10    if (l->tail == da_eliminare) {
11        l->tail = da_eliminare->precedente;
12    }
13
14    // Ricuciamo il nodo precedente
15    if (da_eliminare->precedente != NULL) {
16        da_eliminare->precedente->prossimo = da_eliminare->prossimo;
17    }
18
19    // Ricuciamo il nodo successivo
20    if (da_eliminare->prossimo != NULL) {
21        da_eliminare->prossimo->precedente = da_eliminare->precedente;
22    }
23
24    free(da_eliminare);
25 }
```

16.4.4 I Trade-off Ingegneristici

1. **Consumo di memoria:** Ogni nodo deve memorizzare due puntatori (8 byte ciascuno su sistemi a 64 bit), aumentando l'overhead strutturale.
2. **Complessità del codice:** Ogni inserimento o cancellazione richiede l'aggiornamento simultaneo di un massimo di 4 puntatori, aumentando il rischio di bug e puntatori orfani in RAM.

16.5 Quiz di Autovalutazione

Quiz 1

Domanda

1. Che cos'è un Tipo di Dato Astratto (ADT)?

- A) Una struttura dati non ancora implementata dal compilatore.
- B) Un modello logico definito dalle operazioni consentite e dalle regole d'accesso, indipendente dall'implementazione fisica.
- C) Un tipo di puntatore generico `void*` che può indirizzare qualsiasi area dello Heap.
- D) Una macro definita con `\{\}\#define` per memorizzare variabili intere.

Risposta corretta: B
Spiegazione: Gli ADT definiscono il comportamento logico (l'interfaccia pubblica), nascondendo i dettagli implementativi fisici (l'incapsulamento sotto il cofano).

Soluzione

Quiz 2

Domanda

2. Quale logica regola l'accesso agli elementi in una Pila (Stack)?

- A) FIFO (First-In, First-Out)
- B) LIFO (Last-In, First-Out)
- C) LILO (Last-In, Last-Out)
- D) Accesso casuale tramite indice numerico

Risposta corretta: B
Spiegazione: In uno Stack, l'ultimo elemento inserito (cima) è il primo a essere rimosso, proprio come in una pila fisica di piatti.

Soluzione

Quiz 3

Domanda

3. Nell'implementazione di una Pila tramite lista concatenata semplice, perché facciamo coincidere la cima con la testa della lista?

- A) Per risparmiare memoria escludendo il puntatore NULL.
- B) Per effettuare l'inserimento (push) e la rimozione (pop) in tempo costante $O(1)$ senza scorrere la lista.
- C) Perché la testa della lista risiede nello Stack frame invece che nello Heap.
- D) Per ordinare automaticamente gli elementi inseriti.

Risposta corretta: B
Spiegazione: Operare in testa a una lista semplice richiede solo la modifica locale dei puntatori senza dover scorrere i nodi intermedi.

Soluzione

Quiz 4

Domanda

4. Cosa si intende per "Underflow" di una Pila?

- A) L'esaurimento della memoria fisica Heap a seguito di una `malloc` fallita.
- B) Il tentativo di rimuovere un elemento (`pop`) da una pila che è attualmente vuota.
- C) La perdita del puntatore alla testa della lista.
- D) L'accumulo di record di attivazione causato da ricorsione infinita.

Soluzione
Risposta corretta: B
Spiegazione: L'underflow si verifica quando un'operazione di rimozione fallisce perché la struttura dati è priva di elementi.

Quiz 5

Domanda

5. Quale logica regola l'accesso agli elementi in una Coda (Queue)?

- A) FIFO (First-In, First-Out)
- B) LIFO (Last-In, First-Out)
- C) Accesso casuale indicizzato
- D) Ordinamento alfabetico automatico

Soluzione
Risposta corretta: A
Spiegazione: In una Coda, il primo elemento inserito è il primo a uscire, rispecchiando il comportamento di una fila ordinata di persone.

Quiz 6

Domanda

6. Perché una semplice lista concatenata con il solo puntatore head non è efficiente per implementare una Coda?

- A) Perché l'estrazione richiede di scorrere tutti gli elementi.
- B) Perché l'inserimento in coda alla lista richiede uno scorrimento lineare $O(N)$ di tutti i nodi.
- C) Perché non permette la memorizzazione di dati di tipo intero.
- D) Perché causa costanti errori di segmentation fault a runtime.

Soluzione**Risposta corretta: B**

Spiegazione: Senza un riferimento diretto all'ultimo nodo, per inserire in coda occorre scorrere l'intera lista partendo dalla testa, un'operazione costosa all'aumentare dei nodi.

Quiz 7**Domanda**

7. In una Coda ottimizzata con puntatori head e tail, qual è la complessità dell'operazione di accodamento (enqueue)?

- A) Tempo lineare $O(N)$
- B) Tempo costante $O(1)$
- C) Tempo logaritmico $O(\log N)$
- D) Tempo quadratico $O(N^2)$

Soluzione**Risposta corretta: B**

Spiegazione: Sfruttando il puntatore tail che referencia direttamente l'ultimo nodo, l'aggiungimento del nuovo elemento avviene all'istante senza dover effettuare alcuna scansione.

Quiz 8**Domanda**

8. Che cos'è una Lista Doppia Concatenata?

- A) Una lista che può contenere solo coppie di valori uguali.
- B) Una lista formata da nodi contenenti due puntatori, uno al nodo successivo e uno al nodo precedente.
- C) Una lista memorizzata contemporaneamente in due settori diversi della RAM.
- D) Una lista che supporta due puntatori di testa separati.

Soluzione**Risposta corretta: B**

Spiegazione: La bidirezionalità è consentita dalla presenza del puntatore precedente in ogni nodo, permettendo la scansione a ritroso.

Quiz 9

Domanda

9. Nelle liste doppie, qual è la complessità temporale per eliminare un nodo se possediamo già il suo puntatore esatto?

- A) Tempo lineare $O(N)$
- B) Tempo costante $O(1)$
- C) Tempo quadratico $O(N^2)$
- D) L'operazione non è permessa a runtime

Soluzione
Risposta corretta: B
Spiegazione: Poiché il nodo contiene già gli indirizzi sia del successore che del predecessore, non serve effettuare alcuna scansione della lista per ricucire i collegamenti.

Quiz 10

Domanda

10. Qual è il principale svantaggio delle liste doppie rispetto alle liste semplici?

- A) L'impossibilità di allocarle nello Heap.
- B) Un consumo di memoria maggiore per ogni nodo e una maggiore complessità nella gestione del codice (chirurgia dei puntatori).
- C) La mancanza di supporto per l'allocazione dinamica tramite `malloc`.
- D) La limitazione a contenere unicamente payload di tipo float.

Soluzione
Risposta corretta: B
Spiegazione: I due puntatori per nodo raddoppiano lo spazio di RAM dedicato ai collegamenti, e le operazioni sui puntatori richiedono molta attenzione per non perdere i legami bilaterali.

16.6 Esercizi Pratici Risolti

Esercizio 1: Controllo Pila Vuota

Domanda

Scrivere una funzione in C che verifichi se una pila è vuota.


```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  typedef struct Pila;
10
11 bool is_empty_pila(Pila p) {
12     // La pila è vuota se il puntatore di testa è NULL
13     return (p == NULL);
14 }

```

Soluzione**Esercizio 2: Conteggio Nodi in Pila****Domanda**

Scrivere una funzione che conti gli elementi all'interno di una pila senza distruggerla (senza svuotarla).

```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  typedef struct Pila;
9
10 int conta_elementi_pila(Pila p) {
11     int contatore = 0;
12     Nodo *corrente = p; // Usiamo un puntatore temporaneo per non
13     // modificare la testa originale
14     while (corrente != NULL) {
15         contatore++;
16         corrente = corrente->prossimo;
17     }
18     return contatore;
19 }

```

Soluzione

Esercizio 3: Ispezione in Cima (Peek)**Domanda**

Scrivere una funzione peek che legga il valore in cima alla pila restituendo true se l'operazione ha successo, e false se la pila è vuota.

```
1 #include <stdio.h>
2 #include <stdbool.h>
3
4 typedef struct Nodo {
5     int dato;
6     struct Nodo *prossimo;
7 } Nodo;
8
9 typedef struct Pila {
10     Nodo *pila;
11 } Pila;
12
13 bool peek(Pila p, int *valore) {
14     if (p == NULL) {
15         return false; // Pila vuota
16     }
17     *valore = p->dato; // Assegna il valore in cima
18     return true;
19 }
```

Soluzione**Esercizio 4: Inizializzazione Coda****Domanda**

Implementare la struttura dati di controllo e la funzione per inizializzare una coda dinamica a vuota.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct NodeCoda {
5      int dato;
6      struct NodeCoda *prossimo;
7  } NodeCoda;
8
9  typedef struct {
10     NodeCoda *head;
11     NodeCoda *tail;
12 } Coda;
13
14 void inizializza_coda(Coda *c) {
15     c->head = NULL;
16     c->tail = NULL;
17 }

```

Soluzione

Esercizio 5: Stampa Coda

Domanda

Scrivere una funzione che stampi a schermo tutti gli elementi presenti in una coda, dall'inizio alla fine.

```

1  #include <stdio.h>
2
3  typedef struct NodeCoda {
4      int dato;
5      struct NodeCoda *prossimo;
6  } NodeCoda;
7
8  typedef struct {
9      NodeCoda *head;
10     NodeCoda *tail;
11 } Coda;
12
13 void stampa_coda(const Coda *c) {
14     NodeCoda *corrente = c->head;
15     printf("Inizio Coda -> ");
16     while (corrente != NULL) {
17         printf("[%d] -> ", corrente->dato);
18         corrente = corrente->prossimo;
19     }
20     printf("Fine Coda (NULL)\n");
21 }

```

Soluzione

Esercizio 6: Dimensione Coda**Domanda**

Scrivere una funzione che determini la dimensione corrente (numero di elementi) di una coda.

```

1  #include <stdio.h>
2
3  typedef struct NodeCoda {
4      int dato;
5      struct NodeCoda *prossimo;
6  } NodeCoda;
7
8  typedef struct {
9      NodeCoda *head;
10     NodeCoda *tail;
11 } Coda;
12
13 int dimensione_coda(const Coda *c) {
14     int count = 0;
15     NodeCoda *corrente = c->head;
16     while (corrente != NULL) {
17         count++;
18         corrente = corrente->prossimo;
19     }
20     return count;
21 }
```

Soluzione**Esercizio 7: Ricerca in Lista Doppia****Domanda**

Scrivere una funzione che ricerchi un intero in una lista doppia e restituisca l'indirizzo del nodo se trovato, o NULL altrimenti.

```

1  #include <stdio.h>
2
3  typedef struct Nododoppio {
4      int dato;
5      struct Nododoppio *precedente;
6      struct Nododoppio *prossimo;
7  } Nododoppio;
8
9  typedef struct {
10     Nododoppio *head;
11     Nododoppio *tail;
12 } Listadoppia;
13
14 Nododoppio *cerca_lista_doppia(const Listadoppia *l, int target) {
15     Nododoppio *corrente = l->head;
16     while (corrente != NULL) {
17         if (corrente->dato == target) {
18             return corrente; // Trovato
19         }
20         corrente = corrente->prossimo;
21     }
22     return NULL; // Non trovato
23 }

```

Soluzione**Esercizio 8: Inserimento in Testa Lista Doppia****Domanda**

Scrivere la funzione per inserire un elemento in testa a una lista doppiamente concatenata.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nododoppio {
5      int dato;
6      struct Nododoppio *precedente;
7      struct Nododoppio *prossimo;
8  } Nododoppio;
9
10 typedef struct {
11     Nododoppio *head;
12     Nododoppio *tail;
13 } Listadoppia;
14
15 void inserisci_testa_doppia(Listadoppia *l, int valore) {
16     Nododoppio *nuovo = (Nododoppio*) malloc(sizeof(Nododoppio));
17     if (nuovo == NULL) exit(1);
18     nuovo->dato = valore;
19     nuovo->precedente = NULL;
20     nuovo->prossimo = l->head;
21
22     if (l->head == NULL) {
23         // La lista era vuota
24         l->head = nuovo;
25         l->tail = nuovo;
26     } else {
27         l->tail->prossimo = nuovo;
28         l->head->precedente = nuovo;
29     }
30 }
31

```

Soluzione**Esercizio 9: Inserimento in Coda Lista Doppia****Domanda**

Scrivere la funzione per inserire un elemento in coda a una lista doppiamente concatenata.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nododoppio {
5      int dato;
6      struct Nododoppio *precedente;
7      struct Nododoppio *prossimo;
8  } Nododoppio;
9
10 typedef struct {
11     Nododoppio *head;
12     Nododoppio *tail;
13 } Listadoppia;
14
15 void inserisci_coda_doppia(Listadoppia *l, int valore) {
16     Nododoppio *nuovo = (Nododoppio*) malloc(sizeof(Nododoppio));
17     if (nuovo == NULL) exit(1);
18     nuovo->dato = valore;
19     nuovo->prossimo = NULL;
20     nuovo->precedente = l->tail;
21     if (l->tail != NULL)
22         // La lista era vuota
23         l->tail->prossimo = nuovo;
24     l->tail = nuovo;
25     } else {
26         l->tail = nuovo;
27     }
28     l->tail->prossimo = nuovo;
29     l->tail = nuovo;
30 }
31

```

Soluzione**Esercizio 10: Svuota Pila (Deallocazione)****Domanda**

Scrivere una funzione che svuoti completamente una pila deallocando tutti i nodi occupati nello Heap.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  typedef Nodo* Pila;
10
11 void svuota_pila(Pila *p) {
12     Nodo *corrente = *p;
13     while (corrente != NULL) {
14         Nodo *successore = corrente->prossimo; // Salva il
15         // successivo
16         free(corrente);
17         // Libera il nodo
18         corrente = successore;
19         // Sposta corrente in
20         avanti
21     }
22     *p = NULL; // Imposta la testa a NULL
23 }

```

Soluzione

Lezione A

Guida al Setup dell'Ambiente

Per iniziare a scrivere, compilare e debuggare programmi in C, sono necessari tre componenti fondamentali:

1. **Un compilatore:** traduce il codice sorgente C in linguaggio macchina eseguibile (`gcc` o alternativamente `clang`).
2. **Un debugger:** consente di analizzare l'esecuzione del programma passo-passo ed ispezionare la memoria RAM. Utilizzeremo **lldb**, lo standard moderno universale e pienamente supportato anche su architetture Apple Silicon.
3. **Un editor di codice:** per scrivere ed organizzare i sorgenti. Utilizzeremo lo stesso ambiente su tutte le piattaforme: **Visual Studio Code (VS Code)** con un profilo pre-configurato.

Su tutte le piattaforme utilizzeremo ambienti e toolchain equivalenti:

- Su **Windows**: useremo `gcc` / `clang` ed `lldb` all'interno del sottosistema Linux **WSL (Windows Subsystem for Linux)**.
- Su **macOS**: useremo `clang` ed `lldb`, inclusi nei Command Line Tools di Apple (pienamente compatibili con la sintassi `gcc`).
- Su **Linux**: useremo `gcc` / `clang` ed `lldb`, gli strumenti nativi della distribuzione.

A.1 Installazione della Toolchain (Compilatore e Debugger)

L'installazione varia in base al sistema operativo. Segui la sezione specifica per il tuo computer.

A.1.1 Piattaforma Windows (via WSL)

Per avere un ambiente di sviluppo POSIX/Linux allineato ai laboratori didattici, su Windows utilizzeremo **WSL (Windows Subsystem for Linux)** con distribuzione Ubuntu.

Passo 1: Installare WSL

1. Cerca **PowerShell** o **Prompt dei comandi** nel menu Start di Windows, fai click destro e seleziona **Esegui come amministratore**.
2. Digita il seguente comando e premi Invio:

```
1 wsl --install -d Ubuntu
```

3. Al termine dell'installazione, **riavvia il computer**.
4. Dopo il riavvio, si aprirà automaticamente la finestra di terminale Ubuntu. Attendi l'inizializzazione e crea il tuo **username** e **password** per il sistema Linux.

Passo 2: Installare Compilatori e Debugger in WSL

Nel terminale di Ubuntu, esegui il seguente comando per aggiornare il gestore dei pacchetti ed installare gcc, clang ed lldb:

```
1 sudo apt update
2 sudo apt install build-essential clang lldb -y
```

Passo 3: Verificare l'installazione

Verifica la presenza degli strumenti digitando:

```
1 gcc --version
2 clang --version
3 lldb --version
```

A.1.2 Piattaforma macOS

Su macOS utilizzeremo i compilatori e debugger nativi Apple Clang ed LLDB forniti tramite i Command Line Tools.

Passo 1: Avviare l'installazione dei Command Line Tools

Apri il **Terminale** (cercandolo tramite Spotlight con Cmd + Spazio) e digita:

```
1 xcode-select --install
```

Clicca su **Installa** nella finestra che appare ed accetta i termini di licenza.

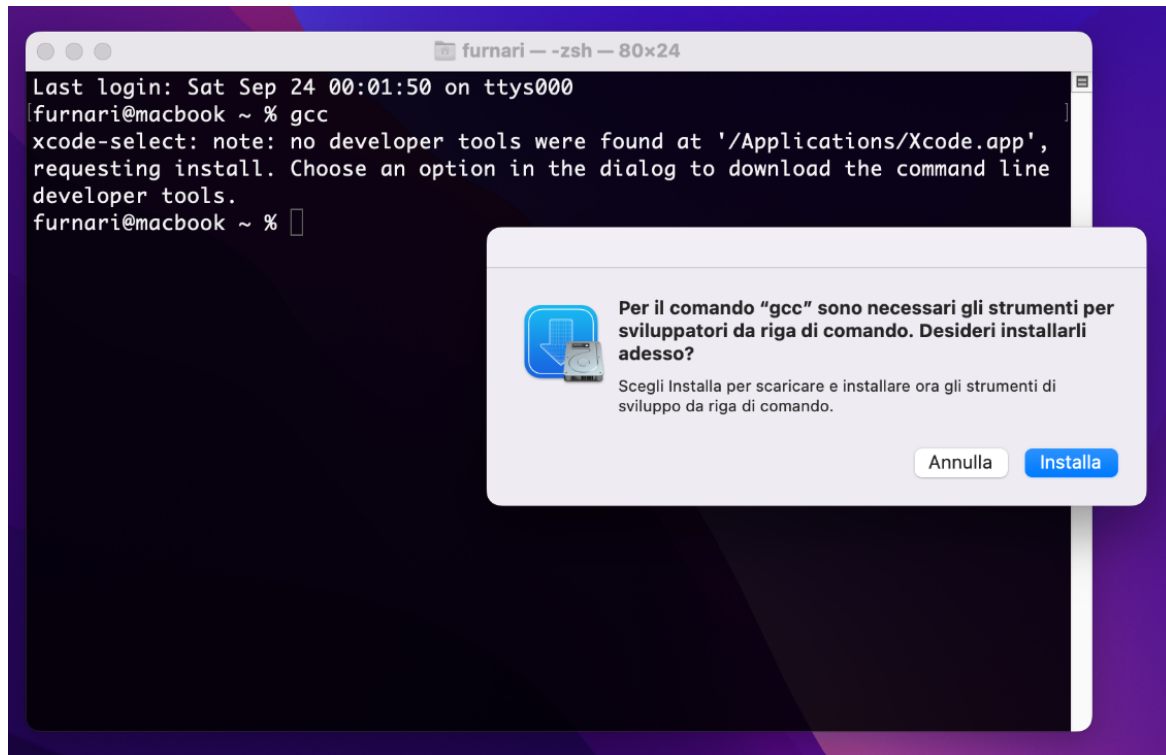


Figura A.1: Richiesta di installazione command line tools

Passo 2: Verificare l'installazione

Attendi il completamento del download. Al termine, verifica la versione installata digitando nel Terminale:

```
1 clang --version
2 lldb --version
```

Suggerimento

Nota su Apple Silicon (M1/M2/M3/M4) Su Mac con processori Apple Silicon, il classico debugger gdb non è più supportato ed instabile. Utilizzeremo esclusivamente **lldb**, che offre prestazioni native straordinarie ed un'integrazione perfetta con VS Code.

A.1.3 Piattaforma Linux (Ubuntu/Debian)

La maggior parte delle distribuzioni Linux consente l'installazione rapida della toolchain completa tramite il gestore di pacchetti.

Passo 1: Installare i pacchetti

Apri il terminale e digita:

```
1 sudo apt update
```

```
2 sudo apt install build-essential clang lldb -y
```

Passo 2: Verificare l'installazione

Verifica la versione dei tools con:

```
1 clang --version
2 lldb --version
```

A.2 Installazione e Configurazione di Visual Studio Code

A.2.1 Passo 1: Download di VS Code

Scarica ed installa l'editor dal sito ufficiale: code.visualstudio.com.

A.2.2 Passo 2: Importazione del Profilo del Corso (Prog1-OZ)

Attenzione

Profilo Obbligatorio per Esercitazioni ed Esami Abbiamo preparato un profilo pre-configurato pronto per essere importato. Questo profilo isola l'ambiente del corso, disabilita l'autocompletamento invasivo (per abituarti a scrivere la sintassi) e installa automaticamente l'estensione per il debugger (CodeLLDB).

1. Scarica il file del profilo da questo link: Prog1-OZ.code-profile.
2. Apri VS Code.
3. Fai clic sull'icona dell'**ingranaggio** in basso a sinistra (Impostazioni) e seleziona **Profiles** -> **Import Profile...**
4. Seleziona il file `Prog1-OZ.code-profile` appena scaricato e clicca su **Create Profile**.
5. Riconoscerai il profilo attivo dall'icona del libro (**Book**) in basso a sinistra.

Nota

Integrazione WSL su Windows Una volta importato il profilo su Windows, apri le estensioni (Ctrl+Shift+X), cerca ed installa l'estensione **WSL** (by Microsoft). Clicca sul pulsante verde in basso a sinistra e seleziona **Connect to WSL** per collegare VS Code al compilatore Linux.

A.3 Test di Compilazione ed Esecuzione da Terminale

Per comprendere il funzionamento reale dei compilatori, non utilizzeremo pulsanti magici o estensioni automatiche. Compileremo direttamente dal terminale integrato.

A.3.1 Passo 1: Scrivere il codice sorgente

Crea un file chiamato `hello.c` ed inserisci il seguente codice:

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("Benvenuti a Programmazione 1!\n");
5     return 0;
6 }
```

A.3.2 Passo 2: Compilare con GCC o Clang

Suggerimento

Prendere buone abitudini: Il flag `-Wall` Quando compilate, aggiungete sempre il flag `-Wall` (*Warnings: All*). Questo comanda a GCC/Clang di segnalare ogni potenziale anomalia o bug non bloccante presente nel vostro codice:

```
1 gcc -Wall hello.c -o hello
```

In alternativa, potete usare Clang:

```
1 clang -Wall hello.c -o hello
```

L'opzione `-o hello` indica al compilatore di generare un file eseguibile chiamato `hello`.

A.3.3 Passo 3: Eseguire il programma

Nel terminale, avvia il programma eseguibile digitando:

```
1 ./hello
```

A schermo apparirà il messaggio di benvenuto corretto!

A.4 Primi Passi con il Debugger `lldb`

Quando il tuo programma presenta bachi o crash di memoria (come i famosi *Segmentation Fault*), il debugger `lldb` ti permette di fermare il tempo ed ispezionare le variabili.

A.4.1 Compilazione per il Debugging (Flag `-g`)

Per consentire a `lldb` di leggere i nomi delle variabili e le righe di codice, occorre compilare aggiungendo il flag `-g` (mantenendo sempre attivo anche `-Wall`):

```
1 clang -g -Wall hello.c -o hello
```

A.4.2 Comandi Fondamentali di lldb

Avvia il debugger passando l'eseguibile:

```
1 lldb ./hello
```

All'interno della shell di lldb (indicata dal prompt (lldb)), puoi usare i seguenti comandi essenziali:

Tabella

Comando	Scorciatoia	Descrizione
breakpoint set -name main	b main	Imposta un punto di interruzione all'inizio della funzione main.
breakpoint set -line 6	b 6	Imposta un punto di interruzione alla riga 6 del sorgente.
run	r	Avvia l'esecuzione del programma fino al primo breakpoint.
next	n	Esegue la riga di codice corrente e passa alla successiva (<i>step over</i>).
step	s	Entra all'interno della funzione chiamata nella riga corrente (<i>step in</i>).
frame variable	v	Stampa il valore di tutte le variabili locali nella funzione corrente.
print mia\{\}_var	p mia\{\}_var	Stampa il valore specifico della variabile mia\{\}_var.
continue	c	Riprende l'esecuzione normale fino al prossimo breakpoint o alla fine.
quit	q	Esci dal debugger lldb.

A.5 Integrazione del Debugger in VS Code (Estensione CodeLLDB)

Sebbene sia possibile fare tutto da riga di comando (consigliato per capire i meccanismi), Visual Studio Code permette di utilizzare un'interfaccia grafica per il debug. Per farlo in modo affidabile (specialmente su macOS e WSL), utilizzeremo l'estensione **CodeLLDB**.

A.5.1 Passo 1: Verificare CodeLLDB in VS Code

Se hai importato il profilo `Prog1-0Z`, CodeLLDB è già installato. Per verificarlo, o per installarlo a mano:

1. Vai nella barra delle estensioni a sinistra (`Ctrl+Shift+X` o `Cmd+Shift+X`).
2. Cerca CodeLLDB (pubblicata da Vadim Chugunov) e, se non è già presente, clicca su **Install**.
3. Se sei su Windows all'interno di WSL, assicurati di installare l'estensione all'interno della sessione di WSL (dovrebbe comparire il pulsante *Install in WSL*).

A.5.2 Passo 2: Configurazione Progetto per Progetto

Il debug e la compilazione grafica in VS Code vanno configurati **progetto per progetto** (ossia per ciascuna cartella aperta). Questo è normale e necessario, poiché progetti diversi potrebbero richiedere comandi di compilazione o parametri di debugging specifici.

La configurazione consiste nel creare una cartella speciale chiamata `.vscode` all'interno della cartella del progetto corrente, contenente due file JSON:

1. **tasks.json**: definisce come compilare il programma prima di avviarlo.
2. **launch.json**: definisce come avviare il debugger collegandolo all'eseguibile compilato.

File `.vscode/tasks.json`

Crea questo file e incolla il seguente contenuto:

```
1 {
2   "version": "2.0.0",
3   "tasks": [
4     {
5       "label": "Prog1 0-Z Simple Build",
6       "type": "shell",
7       "command": "/usr/bin/clang",
8       "args": [
9         "-g",
10        "${file}",
11        "-o",
12        "${fileDirname}/${fileBasenameNoExtension}"
13      ],
14       "group": "build",
15       "problemMatcher": ["$gcc"]
16     }
17   ]
18 }
```

File .vscode/launch.json

Crea questo file e incolla il seguente contenuto:

```
1 {
2   "version": "0.2.0",
3   "configurations": [
4     {
5       "name": "Prog1 0-Z Debugging Profile",
6       "type": "lldb",
7       "request": "launch",
8       "program": "${fileDirname}/${fileBasenameNoExtension}",
9       "args": [],
10      "cwd": "${fileDirname}",
11      "preLaunchTask": "Prog1 0-Z Simple Build"
12    }
13  ]
14 }
```

Nota: Non è fondamentale studiare a memoria i parametri di questi file JSON; il loro scopo è semplicemente compilare il file C attualmente aperto (con l'opzione `-g` attiva) e avviare la sessione di debug grafico.

Suggerimento

Pre-configurati nei Laboratori Nelle attività pratiche e negli assignment scaricabili troverai già la cartella `.vscode` con questi file pre-configurati per facilitarti lo studio. L'uso del debug grafico tramite estensione non è obbligatorio: puoi sempre compilare ed eseguire il debug manuale da terminale con `lldb` come visto nella sezione precedente.

A.5.3 Passo 3: Come usare il Debug Grafico

1. Assicurati di aver aperto la cartella principale del progetto (es. `lab1-demo`) in VS Code.
2. Apri il file `.c` su cui stai lavorando (es. `hello.c` o `orbite.c`).
3. Posiziona il cursore a sinistra del numero di riga in cui vuoi interrompere l'esecuzione e fai clic per impostare un cerchio rosso (**Breakpoint**).
4. Premi **F5** sulla tastiera o vai nel menu a sinistra (icona *Run and Debug*) e fai clic sul pulsante verde *Play* vicino a **Prog1 0-Z Debugging Profile**.
5. VS Code compilerà automaticamente il file C e aprirà una barra dei controlli in alto per permetterti di avanzare passo-passo (Step Over F10, Step In F11, Continue F5), mentre a sinistra potrai visualizzare il valore delle variabili locali in tempo reale.