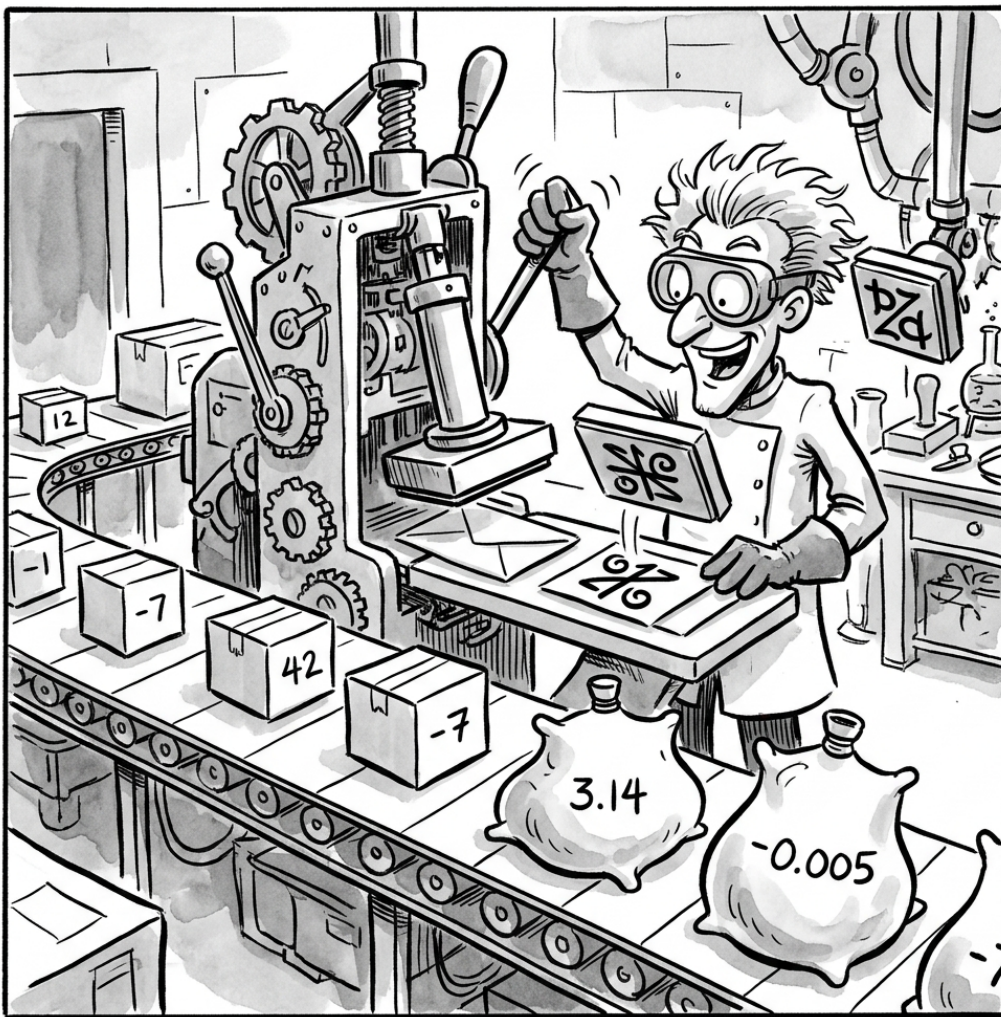


LEZIONE 2

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 1

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`.

1.1 Variabili, Tipi di Dato e I/O Standard

1.1.1 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]`).

1.1.2 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).

1.1.3 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.

1.1.4 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!

1.1.5 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.
```

1.2 Input/Output e Formattazione

1.2.1 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.

1.2.2 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).

1.2.3 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.

1.2.4 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 }
```

1.3 Espressioni, Operatori e Logica

1.3.1 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).

1.3.2 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

1.3.3 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.

1.3.4 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 $x = x - 1$;

1.3.5 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`).

1.4 Logica Booleana e Decisioni Condizionali

1.4.1 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

1.4.2 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 \{\}\&\{\}& B	Risultato A \\{\}\{\} \{\}\{\} B	Risultato !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

1.4.3 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 }
```

1.4.4 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.

1.5 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: 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).

Risposta corretta: B

Soluzione

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: La dichiarazione riserva l'area di memoria ma non la azzera. Leggere una variabile non inizializzata produce comportamenti imprecisati basati sui bit residui in quella cella.

Risposta corretta: C

Soluzione

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.

Soluzione**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`.

Quiz 6**Domanda****6. Quanto restituisce l'espressione aritmetica $5 / 2$ in C?**

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

Soluzione**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.

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;`

Soluzione**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.

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.

1.6 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
16    return 0;
17 }
```

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.