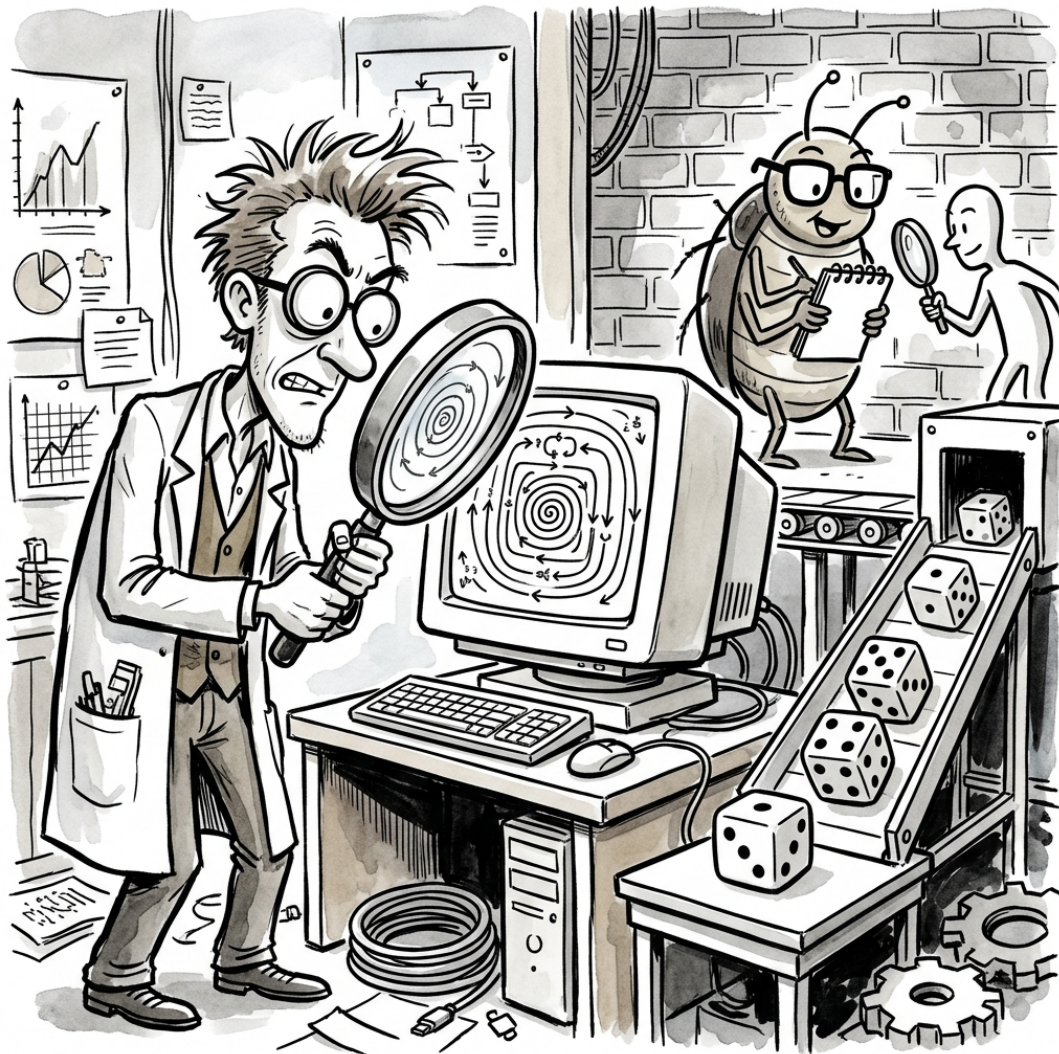


LEZIONE 5

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 1

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.

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

1.1.1 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
6     ↪ 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 }
```

1.1.2 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.

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

1.2.1 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.

1.2.2 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.

1.3 Numeri Pseudo-Casuali e Debugging

1.3.1 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 }

```

1.3.2 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
12         // ↳ a i
13         for (int j = 2; j <= i; j++) { // BUG 3: j <= i includer? anche i
14             // ↳ stesso, quindi i % i == 0 sar? sempre vero!
15             if (i % j == 0) {
16                 is_prime = 0;
17                 break;
18             }
19         }
20
21         if (is_prime) {
22             somma += i;
23             printf("Trovato primo: %d\n", i);
24         }
25     }
26 }
```

```

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).

1.4 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.

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.

Soluzione

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.

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.

Soluzione

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

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

Soluzione

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.

1.5 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     printf("Inserisci un numero intero positivo: ");
6     if (scanf("%d", &n) != 1 || n < 0) {
7         printf("Inserimento non valido.\n");
8         return 1;
9     }
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(" * ");
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 .

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).

Spiegazione dell'algoritmo:

```

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     return 0;
24 }
25
26
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

```