



Programmazione 1 (Canale O-Z)

Lezione 5: Iterazione Avanzata, Visibilità, Casualità e Debugging

Docente: Prof. Antonino Furnari

Email: `antonino.furnari@unict.it`

Sito Web: antoninofurnari.github.io/prog1-2627

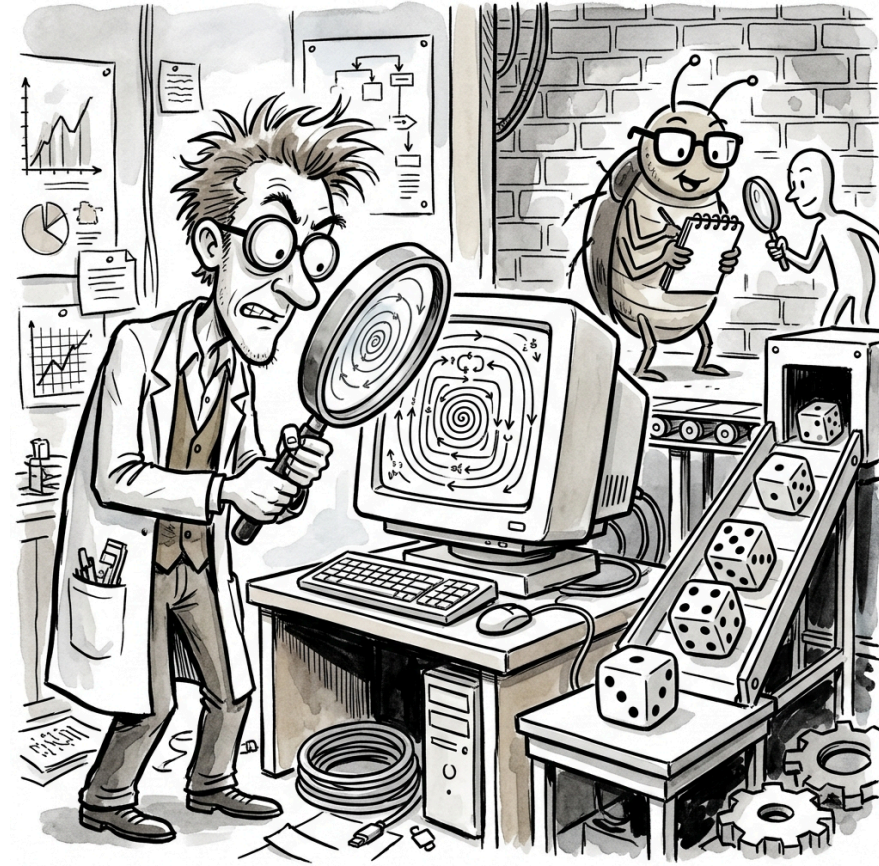
Contare con il `while` : Cosa Serve?

Per contare da 1 a 10 con un `while` servono sempre **tre ingredienti**:

- **Inizializzazione** del contatore;
- **Condizione** controllata in cima;
- **Incremento** in fondo al corpo.

Dimenticarne uno = loop infinito o comportamento imprevisto.

? E se potessimo condensare tutti e tre in un'unica riga?



Il Ciclo `for`: Zuccherò Sintattico

Il `for` non aggiunge potenza di calcolo: condensa in una sola riga i tre elementi essenziali del ciclo contatore.

Ciclo `for`:

```
for (inizializzazione; condizione; incremento) {  
    // corpo  
}
```

Equivalente `while`:

```
inizializzazione;  
while (condizione) {  
    // corpo  
    incremento;  
}
```

- Tutti e tre gli elementi sono visibili in un'unica riga.
- Riduce il rischio di dimenticare l'incremento e creare loop infiniti.

Esempio: Numeri Pari da 2 a 10

Con `for` :

```
#include <stdio.h>

int main(void) {
    int limit = 10;
    for (int i = 2; i ≤ limit; i += 2) {
        printf("%d ", i);
    }
    printf("\n");
    return 0;
}
```

Equivalente con `while` :

```
#include <stdio.h>

int main(void) {
    int limit = 10;
    int i = 2; // 1. Inizializzazione
    while (i ≤ limit) { // 2. Condizione
        printf("%d ", i);
        i += 2; // 3. Incremento
    }
    printf("\n");
    return 0;
}
```

 **Live:** gcc -Wall pari.c -o pari && ./pari ➡ 2 4 6 8 10. Poi cambiare limit e il passo i += 2.

Traccia del `for`: Passo per Passo

```
for (int i = 2; i ≤ limit; i += 2) { printf("%d ", i); } // limit = 10
```

Passo	i	i ≤ 10 ?	Azione	Output finora
Iterazione 1	2	✓	stampa 2, poi i += 2	2
Iterazione 2	4	✓	stampa 4, poi i += 2	2 4
Iterazione 3	6	✓	stampa 6, poi i += 2	2 4 6
Iterazione 4	8	✓	stampa 8, poi i += 2	2 4 6 8
Iterazione 5	10	✓	stampa 10, poi i += 2	2 4 6 8 10
Controllo finale	12	✗	uscita dal ciclo	2 4 6 8 10

- Inizializzazione **una sola volta**; poi condizione → corpo → incremento. ⚠ La `i` di `for (int i ...)` è **locale al ciclo**.

Il Ciclo `do-while` : Post-Condizionato

Con `while` e `for` la condizione si valuta **prima**: se è falsa dall'inizio, il corpo non esegue mai.

Con `do-while` il corpo esegue **almeno una volta**, poi la condizione viene valutata in coda:

```
do {  
    // corpo (eseguito almeno una volta)  
} while (condizione); // ← punto e virgola obbligatorio!
```

? Vogliamo chiedere un voto (18-30) finché l'utente non ne inserisce uno valido: quale ciclo scegliereste?



Validazione dell'Input — Opzione A: `while`

```
#include <stdio.h>

int main(void) {
    int voto = 0; // Dobbiamo inizializzarlo con un valore fittizio per forzare l'ingresso nel loop

    printf("Inserisci il voto dell'esame (18-30): ");
    scanf("%d", &voto);

    while (voto < 18 || voto > 30) {
        printf("Voto non valido! Inserisci il voto dell'esame (18-30): ");
        scanf("%d", &voto); // Duplicazione del codice di lettura
    }

    printf("Voto registrato: %d\n", voto);
    return 0;
}
```

- Funziona, ma la coppia `printf` + `scanf` è **duplicata**: prima del ciclo e dentro il ciclo.
- Se un domani cambiamo il formato di lettura, dobbiamo ricordarci di modificarlo in **due** punti.

Validazione dell'Input — Opzione B: `do-while`

```
#include <stdio.h>

int main(void) {
    int voto; // Nessuna necessità di inizializzazione fittizia

    do {
        printf("Inserisci il voto dell'esame (18-30): ");
        scanf("%d", &voto);

        // Se il voto inserito non è valido, stampiamo un avviso prima di ripetere il ciclo
        if (voto < 18 || voto > 30) {
            printf("Voto non valido! Riprova.\n\n");
        }
    } while (voto < 18 || voto > 30);

    printf("Voto registrato con successo: %d\n", voto);
    return 0;
}
```

- Nessuna duplicazione dell'I/O e messaggio di errore comunque esplicito.

 **Live:** `gcc -Wall voto.c -o voto && ./voto`, inserire 35, poi 10, poi 25.

Cicli Annidati: La Griglia di Coordinate

Per ogni iterazione del ciclo **esterno**, il ciclo **interno** viene eseguito per intero.

```
#include <stdio.h>

int main(void) {
    int righe = 3;
    int colonne = 4;

    for (int r = 1; r ≤ righe; r++) {           // esterno
        for (int c = 1; c ≤ colonne; c++) {     // interno
            printf("(%d,%d) ", r, c);
        }
        printf("\n"); // a capo a fine riga
    }
    return 0;
}
```

Output:

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

? Quante volte viene eseguita la `printf` interna? E la `printf("\n")`?

Scope e Lifetime delle Variabili

Ogni variabile in C possiede:

- uno **Scope**: l'ambito in cui il nome della variabile è **visibile** e utilizzabile;
- un **Lifetime**: il tempo durante il quale la variabile **occupa fisicamente** memoria.

La regola del Block Scope: ogni coppia di graffe `{ ... }` definisce un blocco con un proprio scope locale.

- Una variabile dichiarata in un blocco è visibile solo **in quel blocco** e nei blocchi in esso nidificati.
- All'uscita dal blocco, la variabile **cessa di esistere** e la sua memoria viene rilasciata.

 Cosa succede se proviamo a usare una variabile fuori dal suo blocco?

Esempi di Codice che Non Compilano

Caso 1: dentro un `if`

```
int x = 10;
if (x > 5) {
    int y = 20; // y nasce qui
    printf("y: %d\n", y); // OK
}
// printf("y: %d\n", y);
// ERRORE: 'y' non esiste qui!
```

Caso 2: dentro un ciclo

```
for (int i = 0; i < 5; i++) {
    int quadrato = i * i; // nasce e muore
} // ad ogni iterazione
// printf("%d\n", quadrato); ← ERRORE!
// printf("%d\n", i); ← ERRORE!
```

 **Live:** decommentare una delle righe e compilare con `gcc -Wall scope.c -o scope`: leggere insieme l'errore (`'y' undeclared`). Il compilatore è la prima linea di difesa!



La Memoria RAM: Lo Stack

Quando una funzione viene avviata (`main` incluso), il sistema operativo riserva una zona di memoria temporanea e organizzata: lo **Stack** (pila).

- Le variabili locali vengono **create** in cima allo Stack entrando nel loro blocco `{}`.
- Vengono **eliminate** automaticamente all'uscita dal blocco, liberando la memoria.
- Lo Stack **non pulisce** i vecchi valori: una variabile non inizializzata contiene un **garbage value** (residuo casuale).

⚠ **Inizializza sempre le tue variabili!** Il debugger, a fine lezione, renderà questo errore visibile.

- Studieremo a fondo lo Stack (e lo *Stack Frame*) quando introdurremo le funzioni.

Anticipazione: Variabili Globali

Una variabile dichiarata **fuori da ogni funzione** è **globale**: vive per tutta la durata del programma in una zona di memoria statica ed è leggibile e modificabile da qualsiasi parte del codice.

```
int contatore_globale = 0; // variabile globale

int main(void) {
    contatore_globale = 5; // visibile ovunque nel file
    return 0;
}
```

 **Pessima pratica!** Le variabili globali rendono il codice instabile, difficile da testare e prone a bug logici. Nei laboratori e nelle prove d'esame il loro uso è **vietato**.



Il Determinismo dei Calcolatori

I computer sono macchine **deterministiche**: non possono generare veri numeri casuali, ma sequenze **pseudo-casuali** prodotte da formule matematiche (libreria `<stdlib.h>`).

- `rand()` restituisce un intero tra `0` e `RAND_MAX` (almeno `32767`).
- L'algoritmo parte da un valore di innesco, il **seme (seed)**, che di default vale `1` .
- Stesso seme ➡ **stessa sequenza** a ogni esecuzione!

? Come possiamo ottenere un seme diverso ogni volta che lanciamo il programma?

Inizializzare il Seme: `srand(time(NULL))`

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    srand(time(NULL)); // Inizializza il seme con l'ora corrente (da invocare una sola volta)

    printf("Numero casuale: %d\n", rand());
    return 0;
}
```

- `time(NULL)` (da `<time.h>`) = secondi trascorsi dal 1° gennaio 1970: cambia continuamente.
- `srand` va chiamata **una sola volta**, all'inizio del `main`.

 **Live:** commentare `srand`, compilare con `gcc -Wall casuale.c -o casuale` ed eseguire due volte  stesso numero. Poi ripristinare `srand` e rieseguire  numeri diversi.

Generare Numeri in un Intervallo $[a, b]$

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

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

Perché funziona?

- `rand() % N` produce un valore in $[0, N - 1]$ (i possibili resti della divisione per N).
- Sommando a otteniamo $[a, a + N - 1]$; imponendo $a + N - 1 = b$ si ricava $N = b - a + 1$.

Caso	Formula
Dado (1-6)	<code>1 + rand() % 6</code>
Percentuale (0-100)	<code>rand() % 101</code>
Temperatura (15-35)	<code>15 + rand() % 21</code>



Esempio: Sfida a Dadi su 3 Lanci

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    srand(time(NULL));
    int somma_giocatore = 0;
    int somma_computer = 0;
    printf("≡≡≡ SFIDA A DADI ≡≡≡\n\n");
    for (int i = 1; i ≤ 3; i++) {
        int tiro_giocatore = 1 + rand() % 6; // Dado da 1 a 6
        int tiro_computer = 1 + rand() % 6;  // Dado da 1 a 6
        printf("Lancio %d:\n", i);
        printf("  Giocatore: %d\n", tiro_giocatore);
        printf("  Computer:  %d\n\n", tiro_computer);
        somma_giocatore += tiro_giocatore;
        somma_computer += tiro_computer;
    }
    printf("Totale Giocatore: %d | Totale Computer: %d\n", somma_giocatore, somma_computer);
    if (somma_giocatore > somma_computer) {
        printf("Hai vinto!\n");
    } else if (somma_computer > somma_giocatore) {
        printf("Vince il computer.\n");
    } else {
        printf("Pareggio!\n");
    }
    return 0;
}
```

Sfida a Dadi: Esecuzione

- Ciclo `for` controllato da contatore + casualità **a ogni iterazione** + **accumulatori** per il punteggio.
- L'output cambia a ogni esecuzione: non ha senso riportarlo qui.

 **Live:** `gcc -Wall dadi.c -o dadi && ./dadi`, eseguire più volte e confrontare i risultati. Poi commentare `srand(time(NULL));` : cosa succede ai lanci?

 E se il programma dà un risultato sbagliato e non capiamo perché? Aggiungiamo `printf` a caso?

La Cultura del Debugging

| "Un programmatore esperto non scrive meno bug: li trova più velocemente."

Invece di inserire `printf` a caso, usiamo il **debugger** per **fermare il tempo** ed esaminare lo stato della memoria nel punto esatto in cui il programma si comporta male.

Flusso di lavoro:

- Compila con il flag `-g` (include i simboli di debug)
- Avvia **LLDB** (da terminale) o **VS Code + CodeLLDB** (grafico)
- Imposta **breakpoint** nei punti sospetti
- Avanza passo-passo e **ispeziona le variabili**



Compilazione e Avvio del Debugger

```
# Compilazione con simboli di debug (e sempre con -Wall!)  
clang -g -Wall programma.c -o programma  
  
# Avvio del debugger  
lldb ./programma
```

- `-g` inserisce nell'eseguibile le informazioni che associano le istruzioni macchina alle **righe del sorgente C**.
- Si apre il prompt `(lldb)`, dal quale impartiamo i comandi.



Comandi LLDB Essenziali

Comando	Azione
<code>b main o b 14</code>	Breakpoint al <code>main</code> o alla riga 14
<code>r</code>	Run (avvia l'esecuzione)
<code>n</code>	Next / Step Over (riga successiva, senza entrare nelle funzioni)
<code>s</code>	Step In (entra nella funzione chiamata)
<code>v</code>	Frame variable: mostra tutte le variabili locali attive
<code>p nome_var</code>	Stampa il valore di una variabile specifica
<code>c</code>	Continue (riprende fino al prossimo breakpoint)
<code>q</code>	Quit (esci da LLDB)



Debugging Grafico in VS Code (CodeLLDB)

Con l'estensione **CodeLLDB** e i file `.vscode/tasks.json` + `launch.json` già configurati:

- **Clicca** sul margine sinistro accanto al numero di riga → cerchio rosso (breakpoint).
- Premi **F5** per avviare la sessione di debug.
- Il pannello **Variables** a sinistra mostra il valore delle variabili in tempo reale.
- Avanza con i pulsanti grafici in alto: **Step Over** (**F10**), **Step Into** (**F11**), **Continue** (**F5**).



Live: aprire il progetto in VS Code, mettere un breakpoint e mostrare il pannello Variables sul programma che segue.



Codice Didattico: 3 Bug da Trovare

Somma dei numeri primi fino a 15. Atteso: $2 + 3 + 5 + 7 + 11 + 13 = 41$.

```
#include <stdio.h>
int main(void) {
    int limit = 15;
    int somma;           // BUG 1: accumulatore non inizializzato a 0 → garbage value!
    int is_prime = 1;    // BUG 2: dichiarato qui, ma non resettato a 1 ad ogni iterazione!
    printf("Calcolo la somma dei numeri primi fino a %d...\n", limit);
    for (int i = 2; i ≤ limit; i++) {
        for (int j = 2; j ≤ i; j++) { // BUG 3: j ≤ i include i stesso → i % i = 0 sempre vero!
            if (i % j == 0) {
                is_prime = 0;
                break;
            }
        }
        if (is_prime) {
            somma += i;
            printf("Trovato primo: %d\n", i);
        }
    }
    printf("La somma finale dei numeri primi è: %d (Attesa: 41)\n", somma);
    return 0;
}
```

Debugging — Passo 1: Esecuzione Iniziale

```
clang -g -Wall primi.c -o primi && ./primi
```

```
Calcolo la somma dei numeri primi fino a 15...  
La somma finale dei numeri primi è: 1042779 (Attesa: 41)
```

- Il valore finale è **incoerente** (qui 1042779, ma cambia da macchina a macchina).
- E non compare **nessuna** riga Trovato primo: ... !

? Da dove viene quel numero? E perché nessun numero viene riconosciuto come primo?

Debugging — Passo 2: BUG 1 (Garbage Value)

```
(lldb) b main
(lldb) r
(lldb) n      # avanziamo fino a prima del ciclo for
(lldb) v      # oppure: p somma
```

- `somma` contiene già un **numero casuale**: un residuo dello Stack, mai azzerato.
- Ogni `somma += i` partirebbe da quel valore "sporco".

✓ **Fix:** inizializzare l'accumulatore a zero ➡ `int somma = 0;`

 **Live:** mostrare lo stesso valore nel pannello **Variables** di VS Code.

Debugging — Passo 3: BUG 3 (Limite del Divisore)

Riavviamo il debug (con `somma = 0`) e avanziamo con `n` nella prima iterazione, `i = 2`:

- Nel ciclo interno, per `j = 2` la condizione `j ≤ i` è **vera**.
- Si esegue `i % j == 0`, cioè `2 % 2 == 0` ➡ **vero**: `is_prime = 0`.
- Il 2, che è primo, viene **scartato**! (E lo stesso accade a ogni `i`, perché prima o poi `j = i`.)

✓ **Fix:** testare solo i divisori **minori** del numero ➡ `j < i`

? Con due bug corretti, il programma ora funziona?

Debugging — Passo 4: BUG 2 (Flag Non Resettato)

Riavviamo con le due correzioni ed eseguiamo:

```
Calcolo la somma dei numeri primi fino a 15...  
Trovato primo: 2  
Trovato primo: 3  
La somma finale dei numeri primi è: 5 (Attesa: 41)
```

- Avanzando con `n`: `i = 2` e `i = 3` sono primi; a `i = 4` il ciclo imposta giustamente `is_prime = 0`.
- A `i = 5` ispezioniamo `p is_prime` ➡ è rimasto a `0` dall'iterazione precedente!

✓ **Fix:** dichiarare `int is_prime = 1;` **dentro** il corpo del `for` esterno (reset a ogni iterazione).

Debugging: Il Codice Corretto

```
#include <stdio.h>
int main(void) {
    int limit = 15;
    int somma = 0;
    printf("Calcolo la somma dei numeri primi fino a %d...\n", limit);
    for (int i = 2; i ≤ limit; i++) {
        int is_prime = 1;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                is_prime = 0;
                break;
            }
        }
        if (is_prime) {
            somma += i;
            printf("Trovato primo: %d\n", i);
        }
    }
    printf("La somma finale dei numeri primi è: %d (Attesa: 41)\n", somma);
    return 0;
}
```

 **Live:** ricompilare con `clang -g -Wall`  `primi 2 3 5 7 11 13` e somma 41.

Quale Ciclo Scegliere?

Struttura	Quando usarla
<code>for</code>	Numero di iterazioni noto (contatore)
<code>while</code>	Condizione da verificare prima di ogni iterazione
<code>do-while</code>	Il corpo deve eseguirsi almeno una volta
Cicli annidati	Operazioni su griglie o strutture bidimensionali

✓ Cosa Abbiamo Imparato Oggi

- `for` = `while` con inizializzazione, condizione e incremento **in una riga**.
- `do-while`: corpo eseguito **almeno una volta**, ideale per validare l'input.
- Cicli annidati: il ciclo interno riparte da capo a ogni iterazione dell'esterno.
- **Block scope**: una variabile vive solo nel suo blocco `{ }`, sullo **Stack**. Niente globali!
- `srand(time(NULL))` una volta, poi `a + rand() % (b - a + 1)`.
- Debugging: `clang -g -Wall, llldb, b, r, n, v, p`. **Inizializza sempre le variabili.**

 **Per casa:** esercizi della Lezione 5 sul sito del corso.

Grazie per l'attenzione!

Sito Web del Corso: antoninofurnari.github.io/prog1-2627