



Programmazione 1 (Canale O-Z)

Lezione 4: Strutture Decisionali e l'Iterazione Fondamentale

Docente: Prof. Antonino Furnari

Email: `antonino.furnari@unict.it`

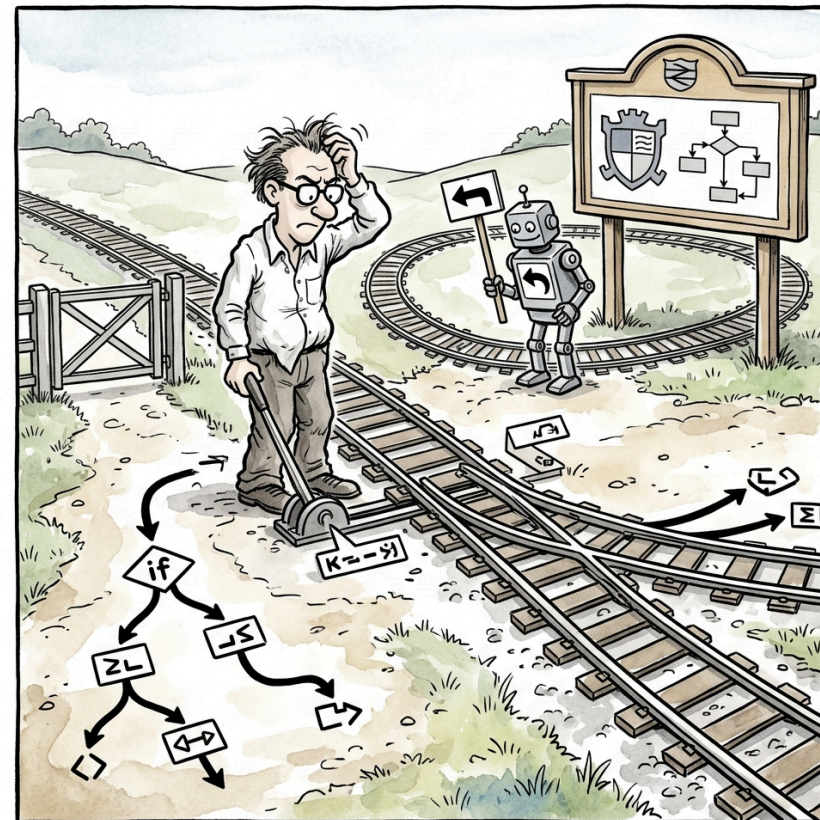
Sito Web: antoninofurnari.github.io/prog1-2627



Oltre il Flusso Lineare

- Finora i nostri programmi sono stati **sequenze lineari** di istruzioni.
- Oggi impariamo ad alterare il flusso per **prendere decisioni** e **ripetere** porzioni di codice.

? Come fa il computer a saltare delle righe di codice o a ripetere un blocco di istruzioni?





Come Ragiona la CPU: I Salti

- La CPU possiede il registro **Program Counter (PC)**: contiene l'indirizzo in RAM dell'istruzione in esecuzione.
- Normalmente il PC avanza in **sequenza**, istruzione dopo istruzione.
- La CPU non conosce blocchi o parentesi graffe: per deviare il flusso esegue **salti (jump)**, caricando un indirizzo diverso nel PC:
 - **Salti condizionati**: saltano solo se una condizione (es. uguaglianza) è soddisfatta.
 - **Salti incondizionati**: saltano sempre.

L' `if` sotto il Cofano: C vs Assembly

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

```
; Assembly concettuale (R1 contiene x)  
cmp R1, 5      ; Confronta R1 con 5  
jne FINE_IF    ; Jump if Not Equal  
mov R1, 10     ; Eseguito solo se R1 == 5  
FINE_IF:  
; Il programma continua...
```

- L' `if` tradotto al contrario: il compilatore genera un salto condizionato (`jne`, *salta se non uguali*) che **evita** il blocco quando la condizione è **falsa**.

 **Live:** incollare il codice C su godbolt.org e cercare nell'Assembly reale la coppia `cmp / jne`.



La Flessibilità del `goto`

Anche in C si possono scrivere salti con `goto` ed **etichette**:

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

Siamo saltati direttamente qui.

- Con `goto` si salta **in avanti** (per saltare codice) ma anche **all'indietro** (creando cicli).

? Cosa succede a un programma pieno di salti avanti e indietro?



Lo Spaghetti Code

- Negli anni '60, l'uso indiscriminato dei salti liberi produceva programmi disordinati: lo **spaghetti code**.
- Salti all'indietro e incroci disordinati ➡ codice **complesso da comprendere e da debuggare**, difficile da mantenere.

? Possiamo fare a meno del `goto` senza perdere potenza? ➡ slide successiva



Il Teorema di Böhm-Jacopini (1966)

- Gli informatici italiani **Corrado Böhm** e **Giuseppe Jacopini** hanno dimostrato che:

Qualsiasi algoritmo computabile può essere espresso usando esclusivamente tre strutture di controllo:

1. **Sequenza**: istruzioni eseguite una dopo l'altra.
 2. **Selezione**: deviazione condizionale del flusso (bivio sì/no).
 3. **Iterazione**: ripetizione di un blocco finché una condizione resta vera.
- **Conseguenza**: possiamo bandire il `goto` **senza perdere potenza computazionale**, affidandoci a strutture pulite e nidificate.



Rappresentazione Grafica: Diagrammi di Flusso

I diagrammi di flusso visualizzano le tre strutture di controllo:

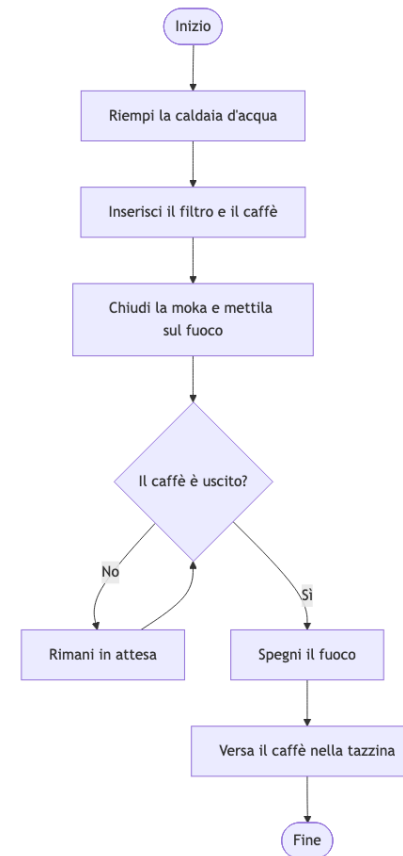
- **Rettangolo:** istruzione sequenziale.
- **Rombo:** selezione condizionale che dirama il flusso (Sì/No).
- **Freccia all'indietro:** ciclo iterativo che torna a un punto precedente.

💡 **Non esagerare con i disegni:** sono utili dal punto di vista concettuale, ma non si usano nella progettazione di software complessi. Basta saperli tracciare a mano libera.



Esempio: Il Diagramma del Caffè

- **Sequenza:** riempi la caldaia, inserisci filtro e caffè, chiudi la moka e mettila sul fuoco.
- **Selezione:** il rombo "Il caffè è uscito?" separa il flusso in Sì/No.
- **Iterazione:** se non è uscito, rimani in attesa e **ripeti** il controllo.
- Quando è uscito: spegni il fuoco e versa il caffè.



Selezione Completa: `if`, `else if`, `else`

? Finora `if` eseguiva un blocco solo se la condizione era vera. E se volessimo una strada alternativa? E se le opzioni fossero tre o più?

```
if (punti ≥ 90) {  
    printf("Eccellente\n");  
} else if (punti ≥ 60) {  
    printf("Sufficiente\n");  
} else {  
    printf("Non sufficiente\n");  
}
```

- Appena una condizione della cascata è vera, la CPU esegue quel blocco e **salta oltre l'intera catena**, ignorando i test successivi.

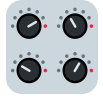
Graffe e Istruzioni Singole

Se il blocco contiene **una sola istruzione**, le graffe `{}` sono opzionali:

```
if (temperatura > 30)
    printf("Fa caldo\n");
else
    printf("Si sta bene\n");
```

- **Buona norma:** inserire **sempre** le graffe, per evitare errori logici quando si aggiungono istruzioni in seguito.

? Se aggiungiamo una seconda istruzione sotto `printf("Fa caldo\n");` senza graffe, cosa succede?



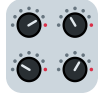
Selezione Multipla Discreta: `switch`

Per confrontare una **variabile discreta** con molti valori fissi, al posto di tanti `else if`:

```
#include <stdio.h>

int main(void) {
    int scelta;
    printf("--- MENU DISTRIBUTORE ---\n");
    printf("1. Caffè (0.50€)\n");
    printf("2. Tè caldo (0.60€)\n");
    printf("3. Cioccolata (0.70€)\n");
    printf("Inserisci la tua scelta (1-3): ");
    scanf("%d", &scelta);
```

```
    switch (scelta) {
        case 1:
            printf("Hai selezionato: Caffè\n");
            break;
        case 2:
            printf("Hai selezionato: Tè caldo\n");
            break;
        case 3:
            printf("Hai selezionato: Cioccolata\n");
            break;
        default:
            printf("Selezione errata!\n");
    }
    return 0;
}
```



switch : Esecuzione

```
gcc -Wall distributore.c -o distributore
./distributore
```

```
--- MENU DISTRIBUTORE ---
1. Caffè (0.50€)
2. Tè caldo (0.60€)
3. Cioccolata (0.70€)
Inserisci la tua scelta (1-3): 2
Hai selezionato: Tè caldo
```

- `default` gestisce ogni valore che non corrisponde a nessun `case` (es. 7 ➡ Selezione errata!).

Il Trabocchetto del Fall-Through

- Senza `break` alla fine di un `case`, la CPU **non si ferma**: esegue a cascata anche le istruzioni dei `case` successivi.
- Può essere sfruttato intenzionalmente (es. stessa azione per `'a'` e `'A'`), ma di solito un `break` mancante è un **bug**.

 **Live:** togliere il `break` del `case 1` e scegliere `1`: compaiono **due** righe, Hai selezionato: Caffè e Hai selezionato: Tè caldo. `-Wall` tace, ma con `gcc -Wall -Wextra` compare `warning: this statement may fall through.`



Vincoli dello `switch`

- Valuta **solo tipi discreti** (`int`, `char`, `short`, `long`): niente stringhe, niente `double`.
- Verifica solo condizioni di **uguaglianza**, non relazioni d'ordine (non si può scrivere `case ≥ 18:`).
- **Non è indispensabile**: ogni `switch` si può riscrivere con `if / else if`. Serve a migliorare la **leggibilità**.

Operatore Ternario (? :) vs if-else

Una selezione rapida per **assegnare un valore** a una variabile:

```
/* Con operatore ternario */  
int max = (a > b) ? a : b;
```

```
/* Con if-else equivalente */  
int max;  
if (a > b) {  
    max = a;  
} else {  
    max = b;  
}
```

- Sintassi: (condizione) ? valore_se_vero : valore_se_falso;

L'Iterazione Fondamentale: Il Ciclo `while`

- Ripete il blocco tra le graffe **finché** la condizione resta vera (diversa da zero).

```
while (condizione) {  
    // Codice da ripetere  
}
```

- **Universalità:** vedremo altri cicli (`for`, `do-while`), ma tutto ciò che si fa con essi si può esprimere con un `while`: è l'**iterazione base universale**.
- **Evitare il loop infinito:** nel corpo deve esserci un'istruzione che modifica le variabili della condizione, rendendola prima o poi **falsa** (`0`).



Divisione per Sottrazioni Successive

Divisione intera **senza** usare l'operatore `/`:

```
#include <stdio.h>

int main(void) {
    int dividendo = 17;
    int divisore = 5;

    int quoziente = 0;
    int resto = dividendo;

    // Sottraiamo finché il resto è maggiore o uguale al divisore
    while (resto ≥ divisore) {
        resto = resto - divisore; // Aggiorna lo stato!
        quoziente++;
    }

    printf("Quoziente: %d, Resto: %d\n", quoziente, resto);
    return 0;
}
```



Traccia del Ciclo (1/4)

Stato iniziale: `quoziente = 0`, `resto = 17`, `divisore = 5`.

Iterazione 1: `resto ≥ divisore` ➡ `17 ≥ 5` ➡ **Vero:** entriamo nel corpo.

Iterazione	resto	quoziente
inizio	17	0
1	12	1



Traccia del Ciclo (2/4)

Iterazione 2: $12 \geq 5$ ➡ Vero.

Iterazione	resto	quoziente
inizio	17	0
1	12	1
2	7	2



Traccia del Ciclo (3/4)

Iterazione 3: $7 \geq 5$ ➡ Vero.

Iterazione	resto	quoziente
inizio	17	0
1	12	1
2	7	2
3	2	3



Traccia del Ciclo (4/4)

Controllo successivo: $2 \geq 5$ ➡ Falso: il ciclo termina.

Quoziente: 3, Resto: 2

- Infatti $17 = 5 \times 3 + 2$.

? Cosa succederebbe se dimenticassimo la riga `resto = resto - divisore;`?

 **Live:** `gcc -Wall divisione.c -o divisione && ./divisione`, poi provare altri valori di `dividendo` e `divisore`.



Massimo e Minimo (Sequenza Fissa)

? Se vi leggesti 10 numeri uno alla volta, come ricordereste il più grande e il più piccolo? Serve ricordarli tutti?

- **Il problema:** trovare il massimo e il minimo tra 10 numeri interi inseriti dall'utente, **senza** memorizzarli tutti.

Valori provvisori:

1. Leggiamo il primo numero e lo impostiamo sia come massimo sia come minimo provvisorio.
2. Per ciascuno dei 9 numeri successivi: se è maggiore del massimo, aggiorniamo il massimo; se è minore del minimo, aggiorniamo il minimo.



Massimo e Minimo: il Codice

```
#include <stdio.h>

int main(void) {
    int numero, max, min;

    printf("Inserisci il 1° numero: ");
    scanf("%d", &numero);
    max = numero; // Inizializzazione provvisoria con il primo elemento
    min = numero;

    int i = 2;
    while (i ≤ 10) { // Ciclo per i restanti 9 numeri
        printf("Inserisci il %d° numero: ", i);
        scanf("%d", &numero);

        if (numero > max) {
            max = numero; // Aggiorna il massimo provvisorio
        }
        if (numero < min) {
            min = numero; // Aggiorna il minimo provvisorio
        }
        i++; // Passa al numero successivo
    }

    printf("\n--- Risultati ---\n");
    printf("Il numero massimo inserito è: %d\n", max);
    printf("Il numero minimo inserito è: %d\n", min);
    return 0;
}
```



Massimo e Minimo: Esecuzione

Invece di digitare 10 numeri, li passiamo tutti insieme con una *pipe*:

```
gcc -Wall maxmin.c -o maxmin  
echo "3 7 -2 9 4 1 8 5 6 0" | ./maxmin
```

```
Inserisci il 1° numero: Inserisci il 2° numero: ... Inserisci il 10° numero:  
--- Risultati ---  
Il numero massimo inserito è: 9  
Il numero minimo inserito è: -2
```

💡 **Perché inizializzare con il primo elemento?** Con `max = 0`, se l'utente inserisse solo numeri **negativi** il massimo resterebbe `0`, che non fa parte della sequenza!

Alterare il Ciclo: `break` e `continue`

- `break` : esce **immediatamente** dal ciclo, saltando alla prima istruzione dopo il blocco.
- `continue` : interrompe l'iterazione corrente e torna subito in cima a **rivalutare la condizione**.

Il problema: leggere i voti d'esame universitari da tastiera e:

1. accettare solo voti tra `18` e `30` (`31` = 30 e lode), saltando gli inserimenti errati con `continue`;
2. fermarsi quando l'utente digita la **sentinella** `-1`, con `break`;
3. stampare alla fine: numero di esami validi, voto massimo, voto minimo e media.



Statistiche Esami (1/2): il Ciclo

```
int voto = 0;
int somma = 0;
int conteggio = 0;
int max = 0; // Il valore più basso possibile per un esame
int min = 100; // Il valore più alto possibile per un esame

printf("Inserisci i voti degli esami (compresi tra 18 e 31; inserisci -1 per terminare):\n");

while (1) { // Ciclo potenzialmente infinito, controllato all'interno
    printf("Inserisci voto: ");
    if (scanf("%d", &voto) != 1) {
        printf("Inserimento non valido.\n");
        break;
    }
    if (voto == -1) {
        break; // Esce immediatamente dal ciclo
    }
    if (voto < 18 || voto > 31) {
        printf("Voto non valido! Deve essere compreso tra 18 e 31 (31 = 30L).\n");
        continue; // Salta il resto del corpo del ciclo e torna in cima
    }
    somma = somma + voto; // Il voto è valido: aggiorniamo le statistiche
    conteggio++;
    if (voto > max) { max = voto; }
    if (voto < min) { min = voto; }
}
```



Statistiche Esami (2/2): i Risultati

```
if (conteggio > 0) {  
    double media = (double)somma / conteggio;  
    printf("\n--- Statistiche Esami ---\n");  
    printf("Esami superati: %d\n", conteggio);  
    printf("Voto massimo: %d\n", max);  
    printf("Voto minimo: %d\n", min);  
    printf("Media voti: %.2f\n", media);  
} else {  
    printf("\nNessun voto valido inserito.\n");  
}
```

- `(double)somma / conteggio`: il **cast** della Lezione 3 evita la divisione intera.

? Qui `max = 0` e `min = 100` funzionano, a differenza dell'esempio precedente: perché?



Statistiche Esami: Esecuzione

```
Inserisci i voti degli esami (compresi tra 18 e 31; inserisci -1 per terminare):  
Inserisci voto: 28  
Inserisci voto: 15  
Voto non valido! Deve essere compreso tra 18 e 31 (31 = 30L).  
Inserisci voto: 30  
Inserisci voto: 31  
Inserisci voto: -1
```

```
--- Statistiche Esami ---  
Esami superati: 3  
Voto massimo: 31  
Voto minimo: 28  
Media voti: 29.67
```



Live: compilare con `gcc -Wall voti.c -o voti`; provare subito `-1` (nessun voto valido) e una lettera al posto di un numero.

✓ Cosa Abbiamo Imparato Oggi

- La CPU devia il flusso con **salti** sul Program Counter; l'`if` diventa un salto condizionato "al contrario".
- Il `goto` porta allo **spaghetti code**; Böhm-Jacopini: bastano **sequenza, selezione, iterazione**.
- Selezione: `if` / `else if` / `else`, `switch` (attenzione al **fall-through**), operatore ternario `? :`.
- Iterazione: `while` è il ciclo universale; aggiornate sempre la condizione.
- Massimo/minimo: inizializzare con il **primo elemento**.
- `break` esce dal ciclo, `continue` salta all'iterazione successiva.

 **Per casa:** esercizi della Lezione 4 sul sito del corso.

Grazie per l'attenzione!

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