UNIMORE

UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

4 – Flusso e Programmazione Strutturata in C

Programmazione 2 - [MN1-1141]

Corso di Laurea in INFORMATICA
Anno accademico 2024/2025

Dr. Alessandro Capotondi
alessandro.capotondi@unimore.it

Risorse di riferimento

Libro di testo “Programmare in C”

- Cap. 4 (sezioni 4.3 – 4.10)

Classi di Istruzioni in C

Supponiamo di voler calcolare (con precisione 10^{-6}):

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

Si chiama "funzione zeta di Riemann".

I costrutti che abbiamo visto finora non sono sufficienti!

- Non sappiamo a priori quante somme dovremo fare
- Non abbiamo un modo per fermarci...
- ...Una volta che sia stata raggiunta la precisione desiderata

Classi di Istruzioni in C

In C esistono due classi di istruzioni:

- Istruzioni semplici (quelle che abbiamo visto finora)
- Istruzioni di controllo (aggregano istruzioni semplici)

Dal punto di vista della sintassi:

```
<istruzione> ::= <istr. semplice> | <istr. di controllo>
<istr. semplice> ::= <espressione>;
```

Esempi:

```
x = 0; y = 1;       // due istruzioni
x = 0, y = 1;       // una istruzione
k++;          // un incremento
;          // istruzione vuota (è valida!)
```

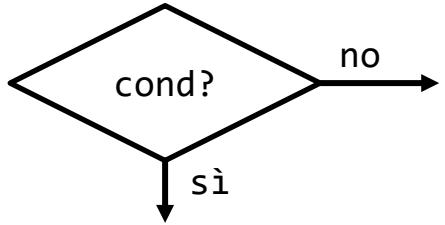
Istruzioni di Controllo

Quali istruzioni di controllo?

Potremmo pensare di ispirarci ai diagrammi di flusso

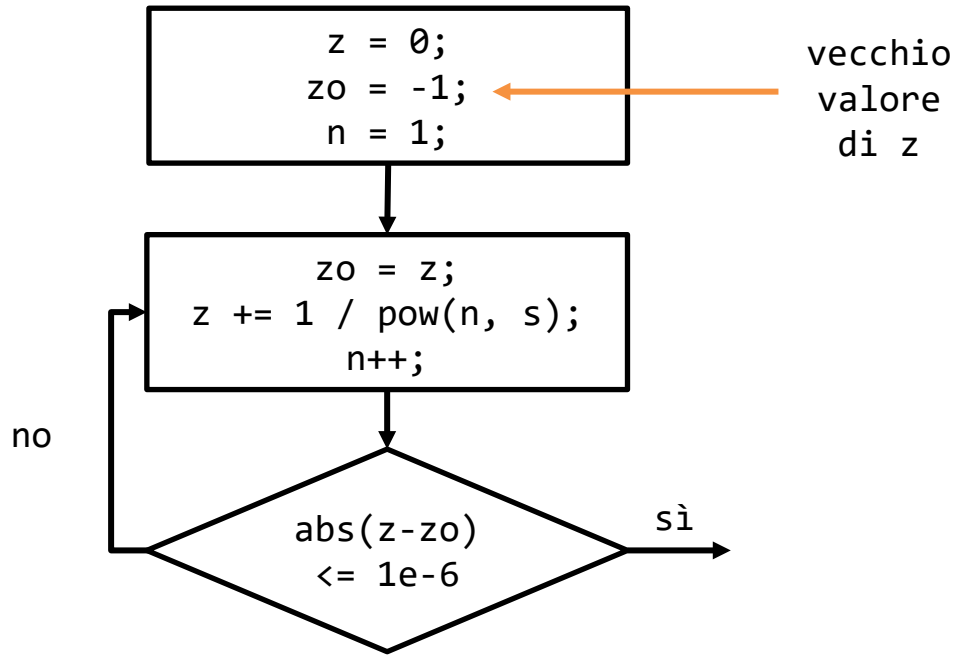
- Le istruzioni compaiono all'interno di "scatole"
- Il flusso di controllo è rappresentato da frecce

Il diagramma può presentare dei "bivi":



- Se la condizione è soddisfatta si prende il ramo "sì"
- Altrimenti si prende il ramo "no"

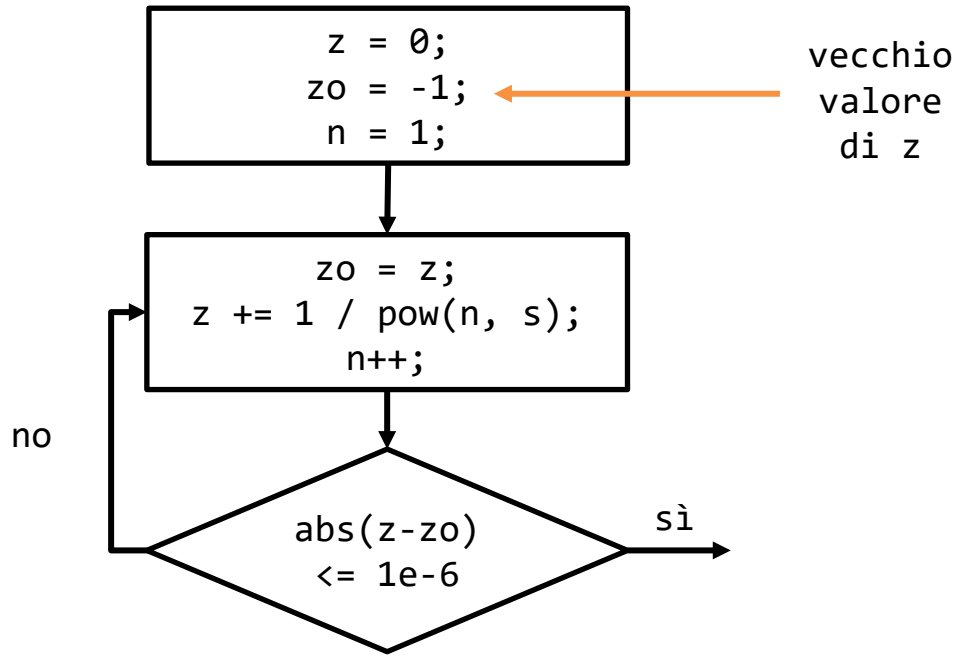
Diagrammi di Flusso



Due modalità di controllo di flusso

- Sequenza (le istruzioni in una scatola si eseguono in fila)
- Salto condizionato (i bivi)

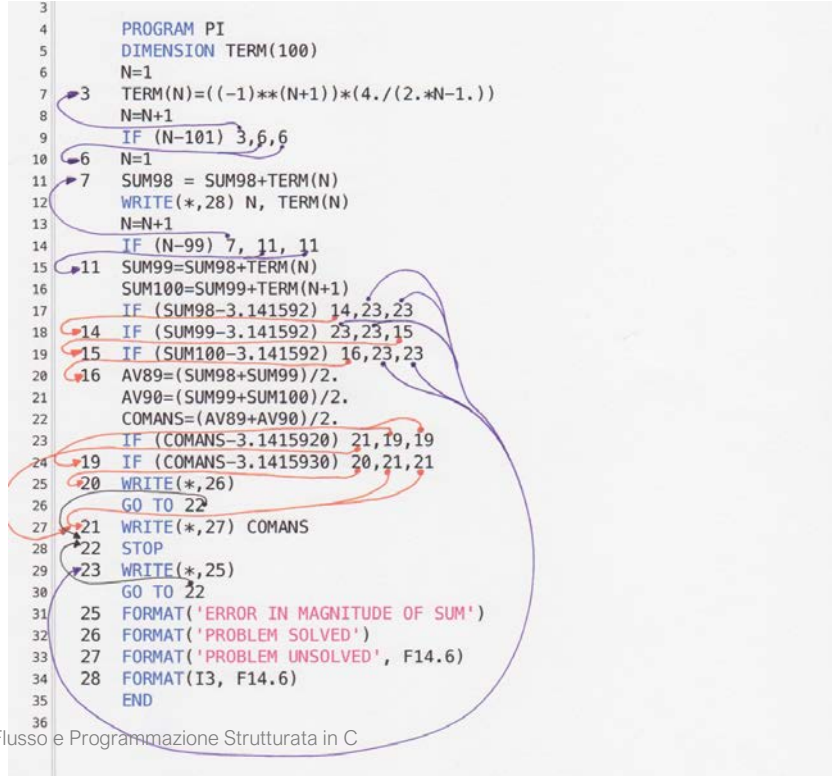
Diagrammi di Flusso



- I diagrammi di flusso erano storicamente molto utilizzati
- Descrivono bene il funzionamento (e.g. del codice assembler)
- Sono semplici da interpretare...

Diagrammi di Flusso

...Finché non capita di dover scrivere un programma complesso



```
      PROGRAM PI
      DIMENSION TERM(100)
      N=1
    3 TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))
      N=N+1
      IF (N-101) 3,6,6
    6 N=1
    7 SUM98 = SUM98+TERM(N)
      WRITE(*,28) N, TERM(N)
      N=N+1
      IF (N-99) 7, 11, 11
   11 SUM99=SUM98+TERM(N)
      SUM100=SUM99+TERM(N+1)
      IF (SUM98-3.141592) 14,23,23
   14 IF (SUM99-3.141592) 23,23,15
   15 IF (SUM100-3.141592) 16,23,23
   16 AV89=(SUM98+SUM99)/2.
      AV90=(SUM99+SUM100)/2.
      COMANS=(AV89+AV90)/2.
      IF (COMANS-3.1415920) 21,19,19
   19 IF (COMANS-3.1415930) 20,21,21
   20 WRITE(*,26)
      GO TO 22
   21 WRITE(*,27) COMANS
   22 STOP
   23 WRITE(*,25)
      GO TO 22
   25 FORMAT('ERROR IN MAGNITUDE OF SUM')
   26 FORMAT('PROBLEM SOLVED')
   27 FORMAT('PROBLEM UNSOLVED', F14.6)
   28 FORMAT(I3, F14.6)
      END
```

Spaghetti Coding

Usando solamente:

- Composizione sequenziale
- Saldo condizionato

Si può scrivere qualunque algoritmo

...Ma è facile produrre codice molto intricato!

Ci si riferisce a questo tipo di risultato come spaghetti coding

- Il rami del flusso di controllo si attorcigliano...
- ...Come in un piatto di spaghetti
- Il codice è corretto, ma poco leggibile e difficile da mantenere

Programmazione Strutturata

Lo spaghetti coding va evitato a tutti i costi

- Negli anni '60 questo problema era molto sentito
- Idea: impedire lo spaghetti coding a livello di linguaggio
- In questo modo sono nati:
  - I linguaggi funzionali (inizialmente LISP)
  - La programmazione strutturata (Edsger W. Dijkstra, 1969)

In programmazione strutturata ci sono tre strutture di controllo:

- Sequenza
- Selezione
- Iterazione

Programmazione Strutturata

Vediamo le idee di base:

Sequenza:

- Le istruzioni vengono eseguite una dopo l'altra
- In C, si usano i blocchi

Selezione

- Viene eseguito un blocco tra tanti, a seconda di una condizione
- In C, si usano le istruzioni `if` e `switch`

Iterazione:

- Un singolo blocco di istruzioni viene eseguito più volte
- In C, si usano le istruzioni `while`, `do..while` e `for`

Blocchi in C

Programmazione Strutturata

La composizione sequenziale è resa in C mediante i blocchi

La sintassi l'abbiamo già vista:

```
<blocco> ::= { { <istruzione> } }
```

- Le istruzioni in un blocco sono eseguite una dopo l'altra
- Le paretensi graffe in rosso sono caratteri necessari
- Non serve il ";" dopo la parentesi graffa chiusa
- È bene che istruzioni nello stesso blocco...
- ...Abbiano la stessa indentazione

Blocchi in C: Esempio

Ogni volta che scriviamo un main definiamo un blocco

```
#include <stdio.h>
int main() {
    int a = 3;
    printf("a = %d\n", a);  // qui b non esiste ancora
    int b = 2;
    printf("a * b = %d\n", a*b); // da qui in poi b esiste
    return 0;
}
```

- Le istruzioni sono eseguite in sequenza...
- ...Quindi le variabili esistono solo dopo la definizione

Blocchi e Variabili

Le variabili definite in un blocco

1) ...Sono utilizzabili solo al suo interno:
   - Il blocco è l'ambito di visibilità (scope) della variabile

2) ...Vengono distrutte al termine del blocco
   - Hanno lo stesso tempo di vita del blocco
   - Distrutte = la memoria viene liberata

Per queste ragioni, tali variabili si dicono locali al blocco

Blocchi e Variabili

Vediamo un esempio con dei blocchi innestati:

```
#include <stdio.h>
int main() {
   int a = 3;
   {
      int b = 2;
   }
   printf("a * b = %d\n", a* b); // ERRORE!
}
```

- Nel blocco esterno a è accessibile (e "viva")
- b è accessibile (e "viva") <u>solo nel blocco interno</u>

Blocchi e Variabili

Vediamo un esempio con dei blocchi innestati:

```
#include <stdio.h>
int main() {
    int a = 3;
    {
        int b = 2;
        printf("a * b = %d\n", a* b); // OK
    }
}
```

- Nel blocco esterno a è accessibile (ed "viva")
- b è accessibile (e "viva") solo nel blocco interno

Località e Nomi di Variabile

Una variabile locale può "mascherare" una variabile esterna

```
#include <stdio.h>
int main() {
   int a = 3;
   {
      int a = 2;
      printf("a = %d\n", a); // a = 2
   }
   printf("a = %d\n", a); // a = 3
}
```

- Ci sono due variabili "a" (una per blocco)
- La variabile "più interna" ha la priorità

Istruzioni Condizionali

Instruzioni di Selezione

Ci sono due istruzioni di selezione (o condizionali) in C

Dal punto di vista della sintassi:

```
<selezione> ::= <scelta> | <scelta multipla>
```

- Basterebbe la prima (che è la più usata)
- La seconda migliora espressività ed efficienza (in certi casi)

Instruzione di Scelta Semplice

Una istruzione di scelta semplice (if) ha la sintassi:

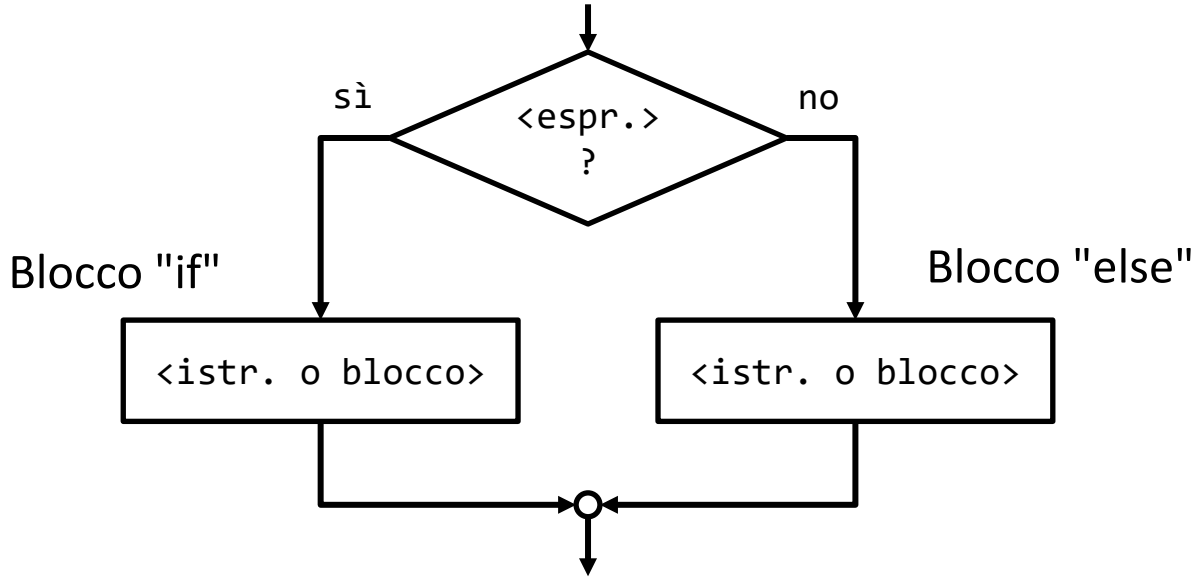
```
<scelta> ::= if (<espr.>) <istr. o blocco>
      [else <istr. of blocco>]
<istr. o blocco> ::= <istruzione> | <blocco>
```

- L'espressione <espr.> viene interpretata come valore logico
  - Viene valutata al momento dell'esecuzione dell'if
- Se <espr.> denota "vero", si esegue la prima istruzione/blocco
- Se <espr.> denota "falso":
  - Se c'è un "else" si esegue la seconda istruzione/blocco
  - Altrimenti non si fa nulla

Instruzione di Scelta Semplice

Si può definire la semantica con un diagramma di flusso

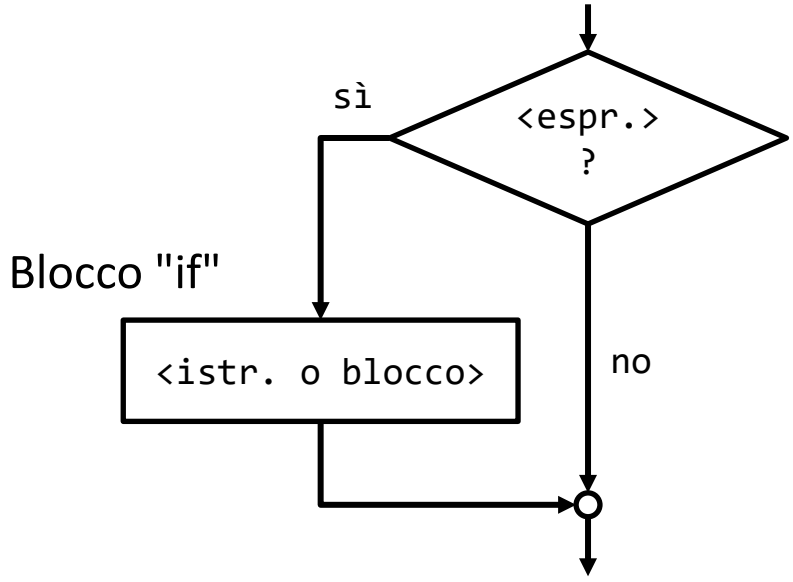
Per un "if" con "else" abbiamo:



Instruzione di Scelta Semplice

Si può definire la semantica con un diagramma di flusso

Per un "if" senza "else" abbiamo:



Instruzione di Scelta Semplice

Vediamo un semplice esempio:

```
if (n > 0) {
    a = b + 5;
    c = a;
}
else n = b;
```

- Abbiamo usato un blocco per la parte "if"
- ...E una singola istruzione per la parte "else"

Instruzione di Scelta Semplice

Stampa del maggiore di due numeri:

```c
#include <stdio.h>
int main() {
   int a = 5, b = 3, max;  // max non inizializzato!
   if (a > b) max = a;
   else max = b;
   printf("max = %d", max);
}
```

- Abbiamo usato un blocco per la parte "if"
- ...E una singola istruzione per la parte "else"

Istruzioni if annidate

Una istruzione potrebbe essere un altro if!

```
if (y != 0)
   if (x > y) z = x / y;
   else z = y / x; // Riferito a if (x > y)
```

- Attenzione ad associare correttamente gli "else"
- Si riferiscono sempre all'istruione "if" più interna
- Suggerimento: usate l'indentazione per essere più chiari

Istruzioni if annidate

Una istruzione potrebbe essere un altro if!

```
if (y != 0)
   { if (x > y) z = x / y; }
   else z = y / x; // Riferito a if (y != 0)
```

- Attenzione ad associare correttamente gli "else"
- Si riferiscono sempre all'istruione "if" più interna
- Suggerimento: usate l'indentazione per essere più chiari

Istruzioni if annidate: Confronto di Orari

Si vogliono confrontare due orari

- Ogni orario è rappresentato con due variabili (ore e minuti):
  - hh1, mm1 e hh2, mm2
- Il risultato del confronto (e.g. cmp) deve valere:
  - -1 se il primo orario precede il secondo
  - 0 se i due orari sono uguali
  - +1 se il primo orario segue il secondo

Esempi:

- hh1 = 12, mm1 = 14, hh2 = 13, mm2 = 5 → cmp = -1
- hh1 = 20, mm1 = 14, hh2 = 13, mm2 = 5 → cmp = 1

Istruzioni if annidate: Confronto di Orari

```
if (hh1 < hh2) cmp = -1;
else
    if (hh1 > hh2) cmp = +1;
    else
        if (mm1 < mm2) cmp = -1;
        else
            if (mm1 > mm2) cmp = +1;
            else cmp = 0;
```

- Lo stesso schema si usa per tutti i confronti lessicografici
- È facile confondersi: usate l'indentazione per chiarire!

Istruzione di Scelta Multipla

In C esiste anche una istruzione di scelta multipla

Esegue istruzioni diverse in base al valore di una espressione

```
<scelta-multipla> ::=
   switch(<espr.>) { <case> {<case>} [<default>] }
<case> ::= case <const>: {<istruzione>}
<default> ::= default: {<istruzione>}
```

- In questo caso <espr.> non è interpretata come valore logico
- Il suo valore viene confrontato con una o più etichette (casi)
- Se il valore corrisponde ad un etichetta...
- ...L'esecuzione salta al "case" corrispondente

Istruzione di Scelta Multipla

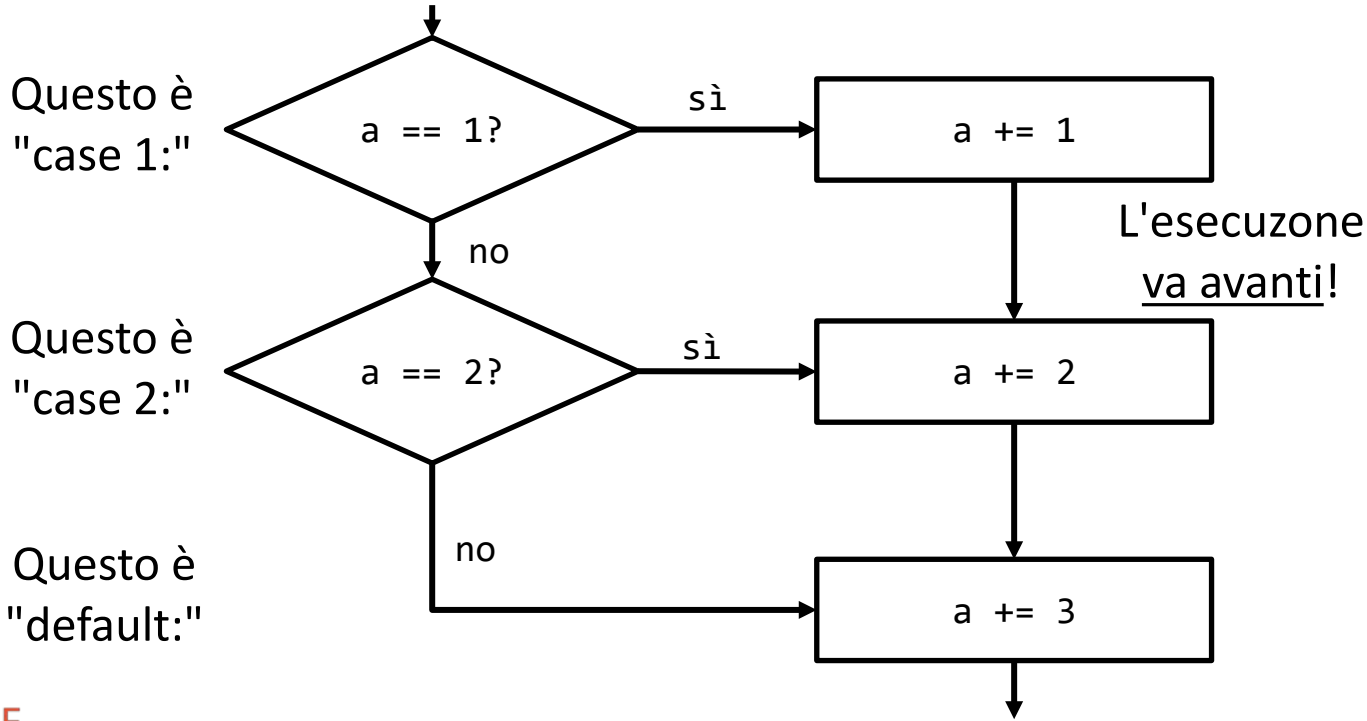
Vediamo un esempio:

```
switch(a) {
    case 1:
        a += 1;
    case 2:
        a += 2;
    default:
        a += 3;
}
```

- La sintassi è piuttosto complessa
- Si raccomanda l'uso dell'indentazione per chiarezza

Istruzione di Scelta Multipla

Semantica con un diagramma di flusso (il nostro esempio):



Istruzione di Scelta Multipla e break

Probabilmente l'aspetto più strano dell'istruzione switch:

```
switch(a) {
    case 1:
        a += 1;
    case 2:
        a += 2;
    default:
        a += 3;
}
```

- Dopo essere salta a (e.g.) "case 1:" l'esecuzione va avanti
- Vengono eseguiti anche "case 2:" e "default"

Istruzione di Scelta Multipla e break

Probabilmente l'aspetto più strano dell'istruzione switch:

```
switch(a) {
    case 1:
        a += 1;
    case 2:
        a += 2;
    default:
        a += 3;
}
```

- Questo aspetto rende "switch" difficile da utilizzare
- Si può ovviare?

Istruzione di Scelta Multipla e break

Si può usare l'istruzione `break`:

```
switch(a) {
    case 1:
        a += 1;
        break;
    case 2:
        a += 2;
        break;
    default:
        a += 3;
}
```

- L'istruzione "break" quando viene eseguita...
- ...Causa la terminazione immediata del blocco

Istruzione di Scelta Multipla e break

Si può usare l'istruzione `break`:

```
switch(a) {
    case 1:
        a += 1;
        break;  // Salta alla fine del blocco
    case 2:
        a += 2;
        break;  // Salta alla fine del blocco
    default:
        a += 3;
}
```

- L'istruzione break quando viene eseguita...
- ...Causa la terminazione immediata del blocco

Istruzione break

L'istruzione break in C:

- Rappresenta una violazione della programmazione strutturata!
  - Termina un blocco prima che sia naturale
- È resa disponibile perché può essere molto utile
  - Nel caso di uno switch è praticamente indispensabile
  - In altri casi può migliorare la leggibilità
  - ...Ma solo se usata con parsimonia

break è come la noce moscata nei tortellini

- Se ne metti un po' vengono meglio
- Se ne metti troppa diventano immangiabili

Un Altro Esempio: Giorni in un Mese


```
switch(mese) {
    case 2:
        if (bisestile) giorni = 29;
        else giorni = 28;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        giorni = 30;
        break;
    default:
        giorni = 31;
}
```

Istruzioni di Iterazione

Istruzioni di Iterazione

In C esistono tre istruzioni di iterazione:

```
<iterazione> ::= <while> | <do while> | <for>
```

- Si chiamano anche "cicli"
- Hanno un unico punto di entrata e uscita nel flusso di controllo
- ...Esattamente come tutte le altre istruzioni
- Ma eseguono ripetutamente una porzione di codice

Si usano molto di frequente

Istruzione while

Iniziamo dal ciclo while:

```
<while> ::= while(<espr.>) <istr. o blocco>
```

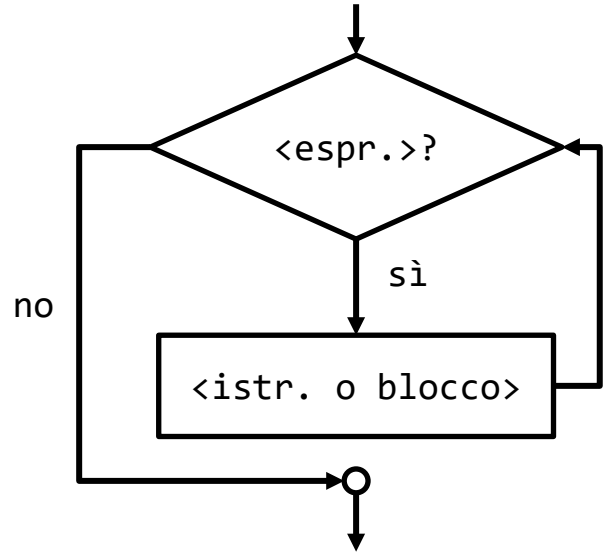
- Si inizia valutando l'espressione <espr.>
- Se <espr.> denota "vero", si esegue l'istruzione/blocco
- Quindi si ricomincia da capo (valutando <espr.>)
- Se <espr.> denota "falso" il ciclo termina, altrimenti si ripete

La condizione/espressione è valutata prima di ogni iterazione

- <istr. o blocco> si dice anche "corpo" del ciclo

Istruzione while

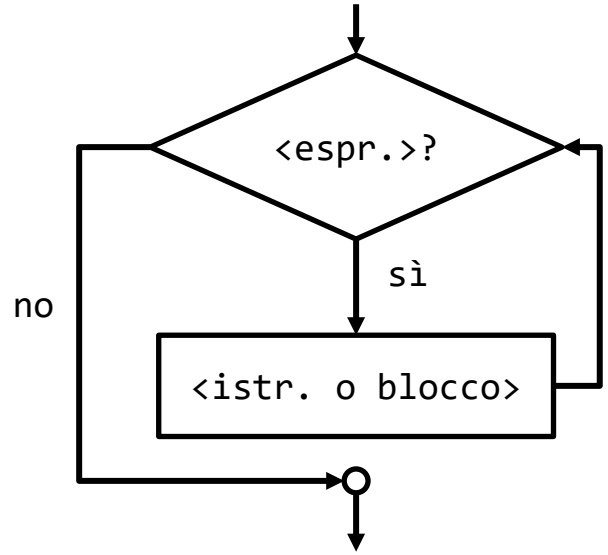
Semantica con un diagramma di flusso



- Il numero di iterazione è in generale indefinito
- Se <espr.> è falsa alla prima valutazione: 0 iterazioni

Istruzione while

Semantica con un diagramma di flusso



- L'istruzione/blocco deve influenzare il risultato di <espr>
- Altrimenti si rischia un ciclo infinito

Esempio

Scrivere un programma che:

- Dati due termini a e b (non negativi)
- Calcoli il prodotto a * b
- ...Come sequenza di somme

Qualche esempio:

- a = 4, b = 3: s = 4 + 4 + 4
- a = 2, b = 5: s = 2 + 2 + 2 + 2 + 2
- a = 2, b = 0: s = 0

Esempio

Una possibile soluzione:

```c
#include <stdio.h>
int main() {
    int a = 4, b = 3;
    int s = 0; // Variabile "accumulatore"
    while (b > 0) {
        s += a;
        b--;
    }
    printf("s = %d\n", s);
}
```

- Ad ogni iterazione sommiamo a ad s e decrementiamo b

Esempio

Scrivere un programma che:

- Dato un numero n (non negativo)
- Calcoli il fattoriale di n, i.e.

$$n! = \prod_{i=1}^{n} i$$

- Si ricordi che 0! = 1

Qualche esempio:

- 4! = 4 * 3 * 2 * 1 = 24
- 6! = 6 * 5 * 4 * 3 * 2 * 1 = 720

Esempio

Una possibile soluzione:

```c
#include <stdio.h>
int main() {
   int n = 6, i = 2;
   int y = 1;  // Inizializzazione del fattoriale
   while (i <= n) {
   y *= i++;
   }
   printf("y = %d\n", y);
   return 0;
}
```

- Aggiornamento di y e i su una singola istruzione

Esempio

Codice errato, con un ciclo infinito

```c
#include <stdio.h>
int main() {
   int n = 6, i = 2;
   int y = 1;
   while (i <= n) {
    y *= i;  // i non viene incrementato
   }
   printf("y = %d\n", y);
   return 0;
}
```

- Potete terminarlo premendo CTRL+C a terminale

Istruzione do...while

Una seconda istruzione di iterazione è data da do...while:

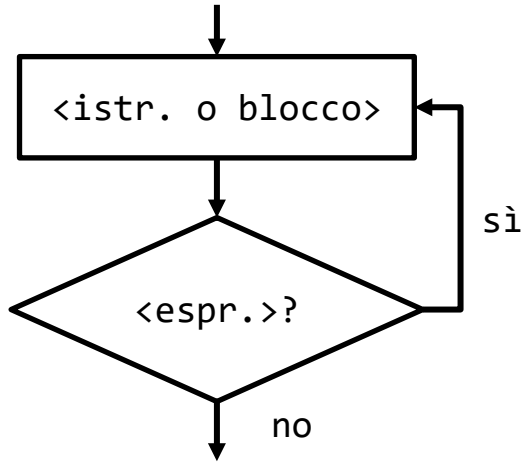
```
<do-while> ::= do <istr. o blocco> while(<espr.>);
```

- Si inizia valutando eseguendo il blocco/istruzione
- La condizione viene valutata dopo ogni iterazione
- Se <espr.> denota "vero", si ripete il processo
- Se <espr.> denota "falso" il ciclo termina

Notate il ";" in fondo (nel while non serve)

Istruzione do...while

Semantica con un diagramma di flusso



- Il numero di iterazione è indefinito (come nel while)
- Se <espr.> è falsa alla prima valutazione: 1 iterazione

Esempio

Scrivere un programma che:

- Dato un numero s (intero non negativo)
- Calcoli la funzione zeta di Riemann:

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

- ...Con una precisione desiderata di 10^{-6}

Come possiamo fare?

- Consideriamo il valore di z prima e dopo un aggiornamento
- Se la distanza è inferiore a 10^{-6}, ci possiamo fermare

Esempio

Una possibile soluzione

```c
#include <stdio.h>
#include <math.h>
int main() {
   double z = 0, zold;
   int s = 2, n = 1;
   do {
    zold = z;  // salvo il vecchio valore di z
    z += 1.0 / pow(n, s);  // aggiorno z
    n++;  // aggiorno n
   } while (fabs(z - zold) > 1e-6);
   printf("z(%d) = %lf\n", s, z);
}
```

Istruzione for

L'instruzione/ciclo for è una evoluzione del while

Mira ad evitare alcuni errori frequenti:

- Mancata inizializzazione di variabili
- Mancato aggiornamento a fine ciclo

È ideata per cicli con un numero di iterazioni noto a priori

- Si può usare anche quando non è vero...
- ...Ma in quel caso un while potrebbe essere più appropriato

Istruzione for

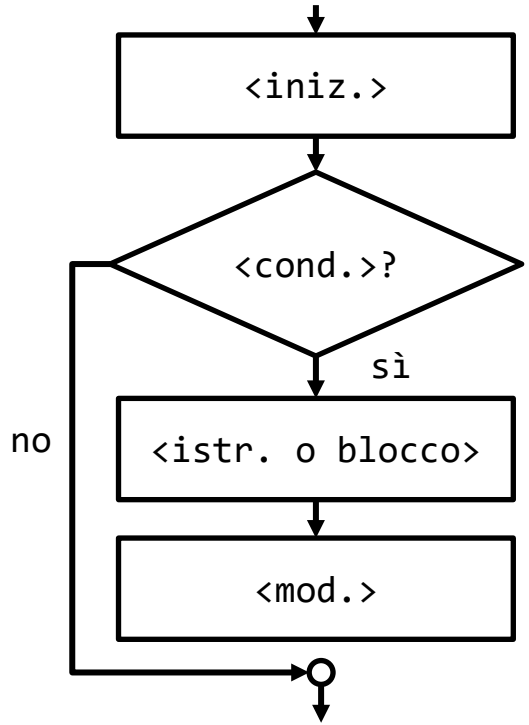
Vediamo la sintassi di una istruzione/ciclo for:

```
<for> ::= for(<iniz.>; <cond.>; <mod.>) <istr. o blocco>
<iniz.> ::= <definizione> | <espressione>
<cond.> ::= <espressione>
<mod.> ::= <espressione>
```

- <iniz.> è una espressione o una definizone di variabile
- <cond.> è una espressione valutata prima di ogni iterazione
- <mod.> è una espressione valutata dopo ogni iterazione

Istruzione for

Vediamo la semantica del for, usando un diagramma di flusso:



Istruzione for: esempi

Qualche piccolo esempio

Stampare 10 volte "ciao":

```
for (i = 0; i < 10; i++) printf("ciao\n");
```

Calcolare 2^{10}:

```
n = 1;
for (i = 0; i < 10; i++) n *= 2;
```

Istruzione for e Definizione di Variabili

L'inizializzazione può contenere una definizione di variabile:

```
for (int i = 0; i < 10; i++) printf("ciao\n");
```

- La variabile i viene definita quando inizia l'esecuzione del for
- È una possibilità offerta dallo standard C99 in poi

Le variabili definite in questo modo sono locali al (blocco del) for

```
n = 1;
for (int i = 0; i < 10; i++) n *= 2;
```

Istruzione for: esempi

Convertiamo il programma per il fattoriale da while a for:

```c
#include <stdio.h>
int main() {
   int n = 6, i = 2;
   int y = 1;
   while (i <= n) {
   y *= i++;
   }
   printf("y = %d\n", y);
   return 0;
}
```

Istruzione for: esempi

Convertiamo il programma per il fattoriale da while a for:

```
#include <stdio.h>
int main() {
   int n = 6, i = 2;
   int y = 1;
   for (;i <= n;) {    // Usiamo for invece di while
   y *= i++;
   }
   printf("y = %d\n", y);
   return 0;
}
```

- L'inizializzazione e modifica possono essere assenti!

Istruzione for: esempi

Convertiamo il programma per il fattoriale da while a for:

```c
#include <stdio.h>
int main() {
   int n = 6, i;
   int y = 1;
   for (i = 2; i <= n;) { // Inizializziamo i nel for
    y *= i++;
   }
   printf("y = %d\n", y);
   return 0;
}
```

- L'inizializzazione nel for diventa più esplicita

Istruzione for: esempi

Convertiamo il programma per il fattoriale da while a for:

```c
#include <stdio.h>
int main() {
   int n = 6, i;
   int y = 1;
   for (i = 2; i <= n; i++) { // Qui l'incremento
   y *= i;
   }
   printf("y = %d\n", y);
   return 0;
}
```

- Spostare l'incremento nel for risulta più chiaro

Istruzione for: esempi

Convertiamo il programma per il fattoriale da while a for:

```
#include <stdio.h>
int main() {
   int n = 6; // la definizione di i non è più qui
   int y = 1;
   for (int i = 2; i <= n; i++) {
   y *= i;
   }
   printf("y = %d\n", y);
   return 0;
}
```

- Possiamo addirittura definire il "contatore" nel for

Istruzione for: esempi

Convertiamo il programma per il fattoriale da while a for:

```
#include <stdio.h>
int main() {
   int n = 6;
   int y = 1;
   for (int i = 2; i <= n; i++) {
   y *= i;
   }
   printf("y = %d\n", y); // i non è accessibile qui
   return 0;
}
```

- Le variabili definite nel for sono <u>locali al suo blocco</u>

Istruzione for: Casi Estremi

Al limite, anche la condizione in un for può essere assente

In questo caso la si intende sempre vera:

```
for(;;) printf("ciao\n"); // Un ciclo infinito
```

- Non è molto utile in pratica

Un for può avere come come corpo l'istruzione vuota

```
for(int n = 6, f = 1; n > 0; f *= n--);
```

- Calcolo del fattoriale in una riga

Esempio

Calcolo della zeta di Riemann con ciclo for

- Dato un numero s (intero non negativo)
- Calcoli la funzione zeta di Riemann:

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

- ...Con una precisione desiderata di 10^{-6}
- ...Si interrompa in ogni caso il calcolo dopo 1000 iterazioni

Come possiamo fare?

Esempio

Una possibile soluzione

```c
#include <stdio.h>
#include <math.h>
int main() {
   double z = 0, zold;
   int s = 2;
   for(int n = 1; n <= 1000; n++) {
      zold = z;
      z += 1.0 / pow(n, s);
      if (fabs(z-zold) <= 1e-6) break;
   }
   printf("z(%d) = %lf\n", s, z);
}
```

Esempio: Riconoscimento di Triangoli

Dati tre valori a ≦ b ≦ c che rappresentano le lunghezze di tre segmenti, valutare se possono essere i tre lati di un triangolo, e se sì deciderne il tipo (scaleno, isoscele, equilatero).

Per avere un triangolo deve valere: c < (a+b)

Rappresentazione delle informazioni:

- la variabile booleana "triangolo" indica se i tre segmenti possono costituire un triangolo
- le variabili booleane scaleno, isoscele e equil indicano il tipo di triangolo

Esempio: Riconoscimento di Triangoli

Specifica:

```
se a+b>c
   triangolo = vero
   se a=b=c
      equil=isoscele=vero  e  scaleno=falso
   altrimenti
      se a=b o b=c o a=c
         isoscele=vero  e  equil=scaleno=falso
      altrimenti
         scaleno=vero  e  equil=isoscele=falso
altrimenti
   triangolo = falso
```

Esempio: Riconoscimento di Triangoli


```
int main (){
   float a=1.5, b=3.0, c=4.0;
   int triangolo, scaleno, isoscele, equil;
   triangolo = (a+b>c);
   if (triangolo) {
      if (a==b && b==c){
         equil=1; isoscele=1; scaleno=0;
      } else {
         if (a==b || b==c || a==c) {
            isoscele=1; scaleno=0; equil=0;
         } else {
            scaleno=1; isoscele==0; equil=0;
         } // if (a==b || b==c || a==c)
    } // if (a == b && b ==c)
   } // if (triangolo)
}
```

Appendice e best practices

Utilizzo di `break`

- Molto spesso, soprattutto agli inizi, una regola che viene data è di ***non utilizzare il costrutto `break`***
  - ***Inserire le condizioni di interruzioni del ciclo in diversi punti può intaccare la sua leggibilità***
  - ***Quasi sempre è possibile inserire le condizioni di controllo nella direttiva che definisce del ciclo***

- Per ora, considerate valida questa regola e provate a non usare mai break. **Tranne che nei *switch-case*.**

Utilizzo di `continue` (extra)

- `Continue` può essere utile per semplificare il codice nel caso di condizioni all'interno del ciclo

```
for(v=0;v<100;v++){
    if(v%3!=0){
        continue;
    }
    ......
    ......
}
```

Ad esempio, in questo caso rendiamo "evidente"

Casi particolari: *cicli infiniti*

- *Il **for** e **while** possono essere utilizzati anche per creare cicli infiniti*

```
for(;;){
}
```

La condizione è opzionale,
se manca il ciclo non termina mai

```
while(1){
}
```

La condizione è obbligatoria,
mettendo 1 è sempre vera

Casi particolari ed errori comuni

- Discutiamo un paio di errori comuni nella scrittura di controlli di flusso in C
  - Utilizzo di operatori di assegnazione (**=**) invece dei controllo di uguaglianza (**==**), spesso a causa di *typo*
  - Overflow e underflow delle variabili nelle operazioni di controllo dei cicli a causa di scelta di *tipi sbagliati*

= e == [1] (avvertenze nel C e in altri linguaggi)

- Principali operatori di verifica delle condizioni
  - < , > , == , != , >= , <=
- Attenzione! Non ci sono dati booleani nativi

  0 è <u>falso</u> **diverso da 0** è <u>vero</u>
- Attenzione! Non confondete

  <u>assegnazione</u> e <u>verifica di uguaglianaza</u>

  = **==**

<u>i due operatori sono completamente diversi!</u>

= e == [2] (avvertenze nel C e in altri linguaggi)

- Ci sono stati ennumerevoli bug causati dall'utilizzo interscambiato di questi due operatori all'interno degli if

- Per capirci: in Python inizialmente non c'era il supporto all'operatore **=** negli **if** proprio per evitare questi bug!

  - È stato aggiunto in Python **3.8** nella PEP 572, ma **tramite un operatore apposito**! (**:=** , detto *walrus operator*)
    - *(curiosità)* e Guido Van Rossum si è dimesso da *benevolent dictator* dopo l'apporvazione di questa PEP

= e == : best practice [1]

- Per evitare di usare per sbaglio l'operatore sbagliato, in caso di controlli con ***literals*** si consiglia spesso di utilizzare questi ultimi nella parte sinistra dell'operazione.

Ad esempio:

```
int a = ... ;
if(a==0){
   ...
}
```

```
int a = ... ;
if(0==a){
   ...
}
```

Questi blocchi di codice sono **identici, MA...**

= e == : best practice [2]

```
int a = ... ;
if(a==0){
   ...
}
```

```
int a = ... ;
if(0==a){
   ...
}
```

Questi blocchi di codice sono **identici, MA...**
se ci sbagliamo a "scrivere" il codice

```
int a = ... ;
if(a=0){
   ...
}
```

Questo compila (e crea un BUG)

```
int a = ... ;
if(0=a){
   ...
}
```

Questo non compila (e ci accorgiamo dell'errore)

Overflow e underflow delle variabili nei cicli [1]

Esempio (tipico) di **integer overflow**

```
int main(){
    char a;
    for(a=0;a<200;a++){}
    printf("Ho finito\n");
    return 0;
}
```

Qual è l'errore?
Qual è l'effetto di questo errore?

Overflow e underflow delle variabili nei cicli [2]

Esempio (tipico) di **integer overflow**

- int main(){
  - char a; for(a=0;a<200;a++){}

Se va la variabile a di tipo char vale 127, quanto fa a+1?
-128

```
    printf("Ho finito\n");
    return 0;
}
```

Ovvero: questo ciclo non termina mai!

Overflow e underflow delle variabili nei cicli [3]

Esempio (tipico) di **integer underflow**

```
int main(){
    unsigned a;
    for(a=9;a>=0;a++){}
    printf("Ho finito\n");
    return 0;
}
```

Il ragionamento è lo stesso appena visto:
se a==-128, allora a-1==127

Goto [1] (extra)

- Esiste un costrutto di salto incondizionato che rappresenta più da vicino il jump presente nei linguaggi di basso livello

ATTENZIONE: è altamente sconsigliato utilizzarlo! In questo corso non dovete usarlo

- Se lavorerete con C, potreste comunque incontrarlo
  - Ad esempio, un uso tipico per cui viene utilizzato è per la gestione degli errori (ci torneremo con un esempio quando faremo le malloc)

Goto [2] (extra)

jump to label 'end'

```
1 int main(){
2     goto end;
3     printf("Non vengo eseguito\n");
4
5 end:
6     printf("Fine del programma\n");
7     return 0;
8 }
```

Il flusso di esecuzione salta direttamente all'istruzione indicata dalla **label (riga 5)**