Istruzioni iterative (o cicliche)

Esempio: Leggi 5 interi, calcolane la somma e stampala.

Variante non accettabile: 5 variabili, 5 istruzioni di lettura, 5 ...

```
int i1, i2, i3, i4, i5;
scanf("%d", &i1):
...
scanf("%d", &i5);
printf("%d", i1 + i2 + i3 + i4 + i5);
```

Variante migliore che utilizza solo 2 variabili:

```
int somma, i;
somma = 0;
scanf("%d", &i);
somma = somma + i;
...                        /* per 5 volte */
scanf("%d", &i);
somma = somma + i;
printf("%d", somma);
```

$\Longrightarrow$ conviene però usare un'**istruzione iterativa**

Le **istruzioni iterative** permettono di ripetere determinate azioni più volte:

- un numero di volte fissato $\Longrightarrow$ **iterazione (o ciclo) definita**

  Esempio:

  for 10 volte
  do fai un giro del parco di corsa

- finchè una condizione rimane vera $\Longrightarrow$ **iterazione (o ciclo) indefinita**

  Esempio:

  while non sei ancora sazio
  do prendi una ciliegia dal piatto
  mangiala

Istruzione `while`

Permette di realizzare l'iterazione in C.

Sintassi:

```
while (espressione)
  istruzione
```

dove

- *espressione* viene detta **condizione** del ciclo
- *istruzione* viene detta **corpo** del ciclo

Semantica:

- viene valutata l'*espressione*
- se è vera si esegue *istruzione* e si torna a valutare *espressione* procedendo così fino a quando *espressione* diventa falsa
- a questo punto si passa all'istruzione successiva

Nota: se *espressione* è già falsa all'inizio, *istruzione* non viene eseguita per niente

Iterazione definita

Esempio: Stampa 100 asterischi.

Si utilizza un **contatore** per contare il numero di asterischi stampati.

algoritmo stampa di 100 asterischi
  inizializza il contatore a 0
  while il contatore è minore di 100
  do stampa un "*"
    incrementa il contatore di 1

Implementazione:

```
int i;
i = 0;
while (i < 100) {
  printf("*");
  i = i + 1;
}
```

$\Longrightarrow$ si parla anche di **ciclo controllato da contatore**

Il contatore viene detto **variabile di controllo** del ciclo.

Esercizio: Leggere 10 interi, calcolarne la somma e stamparla.

algoritmo somma di 10 numeri letti da tastiera
  inizializza la somma a 0
  for 10 volte
  do leggi un intero
    incrementa la somma dell'intero letto

Si utilizza un contatore per contare il numero di interi letti.

Soluzione: file `cicli/somma.c`

Esempio: Leggi 10 interi **positivi** e stampane il massimo.

Si utilizza un **massimo corrente** con il quale si confronta ciascun numero letto.

algoritmo massimo di 10 interi positivi
  inizializza il massimo a 0
  for 10 volte
  do leggi un intero
    if l'intero letto è maggiore del massimo
    then aggiorna il massimo all'intero letto
  stampa il massimo

Implementazione: file `cicli/massimo.c`

Esercizio: Leggere 10 interi (qualunque) e stamparne il massimo.
Soluzione: file `cicli/massimoi.c`

Il limite di conteggio può anche essere letto da tastiera.

Esercizio: Leggere un intero N e stampare i primi N numeri pari.
Soluzione: file `cicli/pari.c`

Operatori di incremento, decremento e assegnazione

Operazioni del tipo `i = i + 1` / `i = i - 1` sono molto comuni. $\Longrightarrow$

- operatore di **incremento**: `++`
- operatore di **decremento**: `--`

In realtà `++` corrisponde a due operatori:

- **postincremento**: `i++`
  - valore dell'espressione è il valore di `i`
  - side-effect: incrementa `i` di 1
- **preincremento**: `++i`
  - valore dell'espressione è il valore di `i+1`
  - side-effect: incrementa `i` di 1

(analogamente per `i--` e `--i`)

Per **side-effect** si intende la modifica del contenuto di una locazione di memoria.

È l'operazione di base nei linguaggi imperativi, nei quali il concetto fondamentale è lo **stato** del programma (dato dal contenuto di tutte le locazioni di memoria).

Operazione di ***assegnazione***

`x = y` è un'espressione

- valore dell'espressione è il valore di `y` (che è un'espressione)
- **side-effect**: assegna alla variabile `x` il valore di `y`

L'operatore "=" è **associativo a destra**.

Esempio: Qual'è il significato di `x = y = 4` ?

È equivalente a: `x = (y = 4)`

- `y = 4` ... espressione di valore 4 con side-effect su `y`
- `x = (y = 4)` ... espressione di valore 4 con ulteriore side-effect su `x`

Le seguenti **espressioni** sono equivalenti:

```
i = i + 1
++i
```

(valore dell'espressione è `i+1`,
come side-effect incrementa `i` di 1)

Le seguenti **istruzioni** sono equivalenti:

```
i = i + 1;
i++;
++i;
```


Nota sull'uso degli operatori di incremento e decremento

Esempio:

	Istruzione	x	y	z
1	`int x, y, z;`	?	?	?
2	`x = 4;`	4	?	?
3	`y = 2;`	4	2	?
4a	`z = (x + 1) + y;`	4	2	7
4b	`z = (x++) + y;`	5	2	6
4c	`z = (++x) + y;`	5	2	7

N.B.: **Non usare mai così!**

In un'istruzione di assegnazione non ci devono essere altri side-effect (oltre a quello dell'operatore di assegnazione) !!!

Riscrivere così: 4b: `z = (x++) + y;` $\Longrightarrow$

```
z = x + y;
x++;
```

4c: `z = (++x) + y;` $\Longrightarrow$

```
x++;
z = x + y;
```

Ordine di valutazione degli operandi

In generale il C **non** stabilisce qual'è l'ordine di valutazione degli operandi nelle espressioni.

Esempio:

```
int x, y, z;
x = 2;
y = 4;
z = x++ + (x * y);
```

Qual'è il valore di `z`?

- se viene valutato prima `x++`: $2 + (3 * 4) = 14$
- se viene valutato prima `x*y`: $(2 * 4) + 2 = 10$

$\Longrightarrow$ Se una variabile compare più volte e inoltre le viene applicato un operatore di incremento (decremento) allora il **risultato dell'espressione è indeterminato**.

Forme abbreviate dell'assegnazione

```
a = a + b;   ⟹   a += b;
a = a - b;   ⟹   a -= b;
a = a * b;   ⟹   a *= b;
a = a / b;   ⟹   a /= b;
a = a % b;   ⟹   a %= b;
```

Inizializzazione di variabili

L'assegnazione di un valore iniziale ad una variabile può essere effettuata contestualmente alla sua dichiarazione $\Longrightarrow$ **inizializzazione di variabile**

Esempio:

```
int a, b = 5, c;
float pi = 3.14152, x;
```

`b` e `pi` sono inizializzate, `a`, `c` e `x` non lo sono

Istruzione `for`

I seguenti elementi sono comuni ai cicli controllati da contatore:

- variabile di controllo (contatore)
- inizializzazione della variabile di controllo
- incremento (decremento) della variabile di controllo ad ogni iterazione
- verifica se si è raggiunto il valore finale della variabile di controllo

Esempio: Stampa i numeri da 1 a 100.

```
int i;                    /* 1 */
i = 1;                    /* 2 */
while (i <= 100) {        /* 4 */
  printf("%d", i);
  i++;                    /* 3 */
}
```

L'istruzione `for` permette di gestire automaticamente questi aspetti:

```
int i;
for (i = 1; i <= 100; i++)
  printf("%d", i);
```

Sintassi:

```
for (espr-1; espr-2; espr-3)
  istruzione
```

dove

- *espr-1* serve a inizializzare la variabile di controllo
- *espr-2* è la verifica di fine ciclo
- *espr-3* serve a incrementare la variabile di controllo
- *istruzione* è il corpo del ciclo

Semantica: l'istruzione `for` di sopra è equivalente a

```
espr-1;
while (espr-2) {
  istruzione
  espr-3;
}
```

(c'è un'eccezione che riguarda l'istruzione `continue`, che però noi non vediamo)

Esempi:

`for (i = 1; i <= 10; i++)`	$\Longrightarrow$	`i`: $1, 2, 3, \ldots, 10$
`for (i = 10; i >= 1; i--)`	$\Longrightarrow$	`i`: $10, 9, 8, \ldots, 2, 1$
`for (i = -4; i <= 4; i += 2)`	$\Longrightarrow$	`i`: $-4, -2, 0, 2, 4$
`for (i = 0; i >= -10; i -= 3)`	$\Longrightarrow$	`i`: $0, -3, -6, -9$

La sintassi del `for` permette che le *espr-i* siano delle espressioni qualsiasi.

Buona norma:

- usare ciascuna *espr-i* in base al significato descritto prima
- non modificare la variabile di controllo nel corpo del ciclo

Ciascuna delle tre *espr-i* puo' anche mancare:

- i "`;`" vanno messi lo stesso
- se manca *espr-2* viene assunto il valore vero
- se manca una delle tre *espr-i* è meglio usare un'istruzione `while`

Esercizio: Riscrivere tutti i programmi con ciclo visti finora utilizzando l'istruzione `for`.

Esercizio: Scrivere un programma che legge un intero N e calcola e stampa il fattoriale di N.

Soluzione: file `cicli/fattiter.c`

Esercizio: Scrivere un programma che legge un intero N ed una sequenza di interi di lunghezza N, e stampa la somma dei positivi e la somma dei negativi nella sequenza.

Soluzione: file `cicli/sompone.c` (senza `for`) e file `cicli/sompone2.c` (con `for`)

Esercizio: Il valore di π può essere calcolato con la serie

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Scrivere un programma che legge un intero N e calcola il valore di π approssimato ai primi N termini della serie.

Iterazione indefinita

In alcuni casi il numero di iterazioni da effettuare non è noto prima di iniziare il ciclo, perché dipende dal verificarsi di una **condizione**.

Esempio: Leggi interi e sommali, fermandoti quando leggi 0.

```
int i, somma = 0;

scanf("%d", &i);
while (i != 0) {
  somma = somma + i;
  scanf("%d", &i);
}
printf("%d", somma);
```

0 gioca il ruolo di **sentinella** $\Longrightarrow$ si parla anche di **ciclo controllato da sentinella**

N.B. la sentinella **non deve essere compresa** tra i dati di ingresso

Esercizio: Leggere una sequenza di interi terminata da 0 e stamparne la lunghezza.

Soluzione: file `cicli/lung1.c`

Istruzione `do-while`

Nell'istruzione `while` la condizione di fine ciclo viene controllata all'inizio di ogni iterazione.

L'istruzione `do-while` è simile all'istruzione `while`, ma la **condizione viene controllata alla fine di ogni iterazione**.

Sintassi:

```
do
  istruzione
while (espressione);
```

Semantica: è equivalente a

```
istruzione
while (espressione) {
  istruzione
}
```


Note sulla sintassi di `do-while`:

- non serve racchiudere il corpo del ciclo tra "`{`" e "`}`"
- c'è un "`;`" dopo "`while` *espressione*"
- per evitare di confondere "`while` *espressione*`;`" con un'istruzione `while` con corpo vuoto conviene scrivere l'istruzione `do-while` in ogni caso come

```
do {
  istruzione
} while (espressione);
```

Esempio: Lunghezza di una sequenza di interi terminata da 0, usando `do-while`.

Implementazione: file `cicli/lung2.c`

Esercizio: Leggere una sequenza di interi terminata da 0 e stampare la somma dei positivi e la somma dei negativi nella sequenza.

Soluzione: file `cicli/sompone3.c`

Esempio: Leggere due interi positivi e calcolare il **massimo comun divisore**.

Es.: $MCD(12,8) = 4$
$MCD(12,6) = 6$
$MCD(12,7) = 1$

1) Sfruttando direttamente la definizione di MCD

- osservazione: $1 \leq MCD(m,n) \leq min(m,n)$
  $\Longrightarrow$ si provano i numeri compresi tra 1 e $min(m,n)$
- conviene iniziare da $min(m,n)$ e scendere verso 1

algoritmo stampa massimo comun divisore di due interi positivi letti da tastiera
  leggi *m* ed *n*
  inizializza *mcd* al minimo tra *m* ed *n*
  while *mcd* > 1 e non si è trovato un divisore comune
  do if *mcd* divide sia *m* che *n*
    then si è trovato un divisore comune
    else decrementa *mcd* di 1
  stampa *mcd*

Osservazioni:

- il ciclo termina sempre perché ad ogni iterazione
  - o si è trovato un divisore
  - o si decrementa *mcd* di 1 (al più si arriva ad 1)
- per verificare se si è trovato il MCD si utilizza una variabile booleana (usata nella condizione del ciclo)

Implementazione: file `cicli/mcd1.c`

Quante volte viene eseguito il ciclo?

- caso migliore: 1 volta (quando m divide n o viceversa)
  p.es. $MCD(500,1000)$
- caso peggiore: $min(m,n)$ volte (quando $MCD(m,n) = 1$)
  p.es. $MCD(500,1001)$

$\Longrightarrow$ algoritmo si comporta male se m e n sono grandi e $MCD(m,n)$ è piccolo

2) Metodo di Euclide per il calcolo del massimo comun divisore

Permette di ridursi più velocemente a numeri più piccoli, sfruttando la seguente proprietà:

$$MCD(m,n) = \begin{cases} m \quad (\text{o } n), & \text{se } m = n \\ MCD(m-n,n), & \text{se } m > n \\ MCD(m,n-m), & \text{se } m < n \end{cases}$$

Dimostrazione: per **esercizio**
(mostrando che i divisore comuni di m ed n, con $m > n$, sono anche divisori di $m - n$)

Es.: $MCD(12,8) = MCD(12-8,8) = MCD(4,8-4) = 4$

Come si ottiene un algoritmo?
Si applica ripetutamente il procedimento fino a che non si ottiene che $m = n$.

Es.:

m	n	$maggiore - minore$
210	63	147
147	63	84
84	63	21
21	63	42
21	42	21
21	21	$\Longrightarrow MCD(21,21) = MCD(21,42) = \cdots = MCD(210,63)$

algoritmo di Euclide per il MCD di due interi positivi
leggi *m* ed *n*
while *m* $\neq$ *n*
do sostituisci il maggiore tra *m* ed *n* con la differenza tra il maggiore ed il minore
stampa *m* (oppure *n*)

Implementazione: file `cicli/mcd2.c`

Cosa succede se $m = n = 0$? $\Longrightarrow$ il risultato è 0

E se $m = 0$ e $n \neq 0$ (o viceversa)? $\Longrightarrow$ si entra in **un ciclo infinito**

Se si vuole tenere conto del fatto che l'utente possa immettere una qualsiasi coppia di interi, è necessario inserire una verifica sui dati in ingresso.

Usiamo un ciclo di lettura e verifica dei dati in ingresso (fa uso di `do-while`)
Implementazione: file `cicli/mcd3.c`

3) Metodo di Euclide con i resti per il calcolo del massimo comun divisore

Cosa succede se $m >> n$?

Es.: $MCD(1000, 2)$

1000	2
998	2
996	2
...	
2	2

$MCD(1001, 500)$

1001	500
501	500
1	500
...	
1	1

Come possiamo comprimere questa lunga sequenza di sottrazioni?

Quello che in fondo si calcola è il resto della divisione intera. $\Longrightarrow$

Metodo di Euclide: sia $m = n \cdot k + r$ (con $0 \leq r < m$)

$$MCD(m, n) = \begin{cases} n, & \text{se } r = 0 \quad \text{(ovvero, } m \text{ è multiplo di } n\text{)} \\ MCD(r, n), & \text{se } r \neq 0 \end{cases}$$

algoritmo di Euclide con i resti per il calcolo del MCD
leggi *m* ed *n*
while *m* ed *n* sono entrambi $\neq 0$
do sostituisci il maggiore tra *m* ed *n* con
il resto della divisione del maggiore per il minore
stampa il numero tra i due che è diverso da 0

Implementazione: per **esercizio**: file `cicli/mcd4.c`

Esempio: Leggere da input una sequenza di 0 e 1 (separati da spazi), terminata da 2, e calcolare la lunghezza della più lunga sottosequenza di soli 0.

Es.: $0\ 0\ 1\ \underbrace{0\ 0\ 0}\ 1\ 1\ 1\ 1\ 0\ 0\ 2$ $\Longrightarrow$ stampa 3

Variabili utilizzate:
bit ... valore letto
cont ... lunghezza sequenza corrente
maxlung ... lunghezza massima sequenza di soli 0 (temporanea)

algoritmo lunghezza massima sottosequenza di soli 0
inizializza *cont* e *maxlung* a 0
do leggi un *bit*
if *bit* è uguale a 0
then incrementa *cont* di 1
if *cont* $>$ *maxlung*
then poni *maxlung* pari a *cont* (oppure: incrementa *maxlung* di 1)
else poni *cont* pari a 0
while *bit* è diverso da 2
stampa *maxlung*

Implementazione: file `cicli/sequenz1.c`

Esercizio: Migliorare l'algoritmo (e l'implementazione) in modo da aggiornare *maxlung* solo al termine di una nuova sequenza di 0 di lunghezza maggiore delle precedenti.

Soluzione: file `cicli/sequenz2.c`

Cicli annidati

Il corpo di un ciclo può contenere a sua volta un ciclo.

Esempio: Stampa della tavola pitagorica.

algoritmo stampa della tavola pitagorica
  for ogni *riga* tra 1 e 10
  do for ogni *colonna* tra 1 e 10
    do stampa *riga* $*$ *colonna*
    stampa un a capo

Implementazione: file `cicli/pitagor1.c`

```
int riga, colonna;
int Nmax = 10;         /* indica il numero di righe e di colonne */

for (riga = 1; riga <= Nmax; riga++) {
  for (colonna = 1; colonna <= Nmax; colonna++)
    printf("%d ", riga * colonna);
  printf("\n");
}
```


Direttiva di compilazione `#define`

Nel programma precedente, `Nmax` non viene mai modificato (è uguale a 10), tuttavia abbiamo allocato una variabile.

Si può evitare?

- usiamo esplicitamente 10 nel programma? **NO!**
  - in un programma complesso non sappiamo più quale'è il significato di 10 (**magic number**)
  - se vogliamo modificare il valore 10 (ad es. in 15) dobbiamo farlo in molti punti del programma
- definiamo un **identificatore costante**

  `#define Nmax 10`
  - `#define` è una **direttiva di compilazione**
  - dice al compilatore di sostituire ogni occorrenza di `Nmax` con 10 prima di compilare il programma

Output formattato

`printf("x%4dy", 10);` stampa 10 su un campo di ampiezza 4, allineato a destra

```
x  10y#
```

`printf("x%-4dy", 10);` stampa 10 su un campo di ampiezza 4, allineato a sinistra

```
x10  y#
```

Esercizio: Stampare la tavola pitagorica.

1. aggiungendo la riga e la colonna di intestazione della tabella, e
2. usando output formattato per allineare le colonne

Soluzione: file `cicli/pitagor2.c`

`printf("x%6.3gy", 1.238);` stampa 1.238 su un campo di ampiezza 6, arrotondato a 3 cifre significative, allineato a destra

```
x  1.24y#
```


Il numero di iterazioni del ciclo più interno può dipendere dall'iterazione del ciclo più esterno.

Esempio: Stampa una piramide di asterischi di altezza letta in input.

Es.: con *altezza* 4:

```
   *
  ***
 *****
*******
```

riga	blank	*
1	3	1
2	2	3
3	1	5
4	0	7

$\Longrightarrow$ stampa: $(\mathit{altezza} - \mathit{riga})$ blank $(2 \cdot \mathit{riga} - 1)$ asterischi

algoritmo stampa piramide di asterischi
  leggi *altezza*
  for *riga* che va da 1 ad *altezza*
  do stampa $(\mathit{altezza} - \mathit{riga})$ spazi bianchi
    stampa $(2 \cdot \mathit{riga} - 1)$ asterischi
    vai a capo

Implementazione: file `cicli/piramid1.c`

Esercizio: Scrivere un programma che stampa una piramide di numeri (di altezza ≤ 9).

Es.: con altezza 4:

```
   1
  121
 12321
1234321
```

Soluzione: file `cicli/piramid2.c`

Esercizio: Scrivere un programma che legge un intero N e stampa il fattoriale di tutti i numeri compresi tra 1 ed N.

Soluzione: file `cicli/fatttab.c`

Istruzione `break`

Abbiamo visto che l'istruzione `break` permette di uscire da una istruzione `switch`.

In generale, `break` permette di uscire prematuramente da un'istruzione `switch`, `while`, `for` o `do-while`.

Esempio:

```
float a;
int i;

for (i = 0; i < 10; i++) {
  scanf("%g", &a);
  if (a >= 0.0)
    printf("%g\n", sqrt(a));
  else {
    printf("Errore\n");
    break;
  }
}
```

N.B. L'esecuzione di un `break` fa uscire di un solo livello.

`break` altera il flusso di controllo. $\Longrightarrow$ Quando viene usata nei cicli:

- si perde la strutturazione del programma
- si guadagna in efficienza rispetto ad implementare lo stesso comportamento in modo strutturato

Esempio: Codice precedente senza `break`:

```
float a;
int i;
int errore = 0;

for (i = 0; (i < 10) && !errore; i++) {
  scanf("%g", &a);
  if (a >= 0.0)
    printf("%g\n", sqrt(a));
  else {
    printf("Errore\n");
    errore = 1;
  }
}
```