

Iterazione determinata e indeterminata

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

- ▶ un numero di volte fissato \implies **iterazione determinata**

Esempio:

fai un giro del parco di corsa per 10 volte

- ▶ finchè una condizione rimane vera \implies **iterazione indeterminata**

Esempio:

finche' non sei sazio

prendi una ciliegia dal piatto e mangiala

Istruzione `while`

Permette di realizzare l'iterazione in C.

Sintassi:

```
while (espressione)
    istruzione
```

- ▶ `espressione` è la **guardia** del ciclo
- ▶ `istruzione` è il **corpo** del ciclo (può essere un blocco)

Semantica:

1. viene valutata l'`espressione`
 2. se è vera si esegue `istruzione` e si torna ad eseguire l'intero `while`
 3. se è falsa si termina l'esecuzione del `while`
- ▶ Nota: se `espressione` è falsa all'inizio, il ciclo non fa nulla.

Iterazione determinata

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

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

```
int conta, dato, somma;
printf("Immetti 10 interi: ");
somma = 0;
conta = 0;
while (conta < 10) {
    scanf("%d", &dato);
    somma = somma + dato;
    conta = conta + 1;
}
printf("La somma e' %d\n", somma);
```

Esempio: Leggere un intero N seguito da N interi e calcolare la somma di questi ultimi.

- ▶ Simile al precedente: il numero di ripetizioni necessarie non è noto al momento della scrittura del programma ma lo è al momento dell'esecuzione del ciclo.

```
int lung, conta, dato, somma;
printf("Immetti la lunghezza della sequenza ");
printf("seguita dagli elementi della stessa: ");
scanf("%d", &lung);
somma = 0;
conta = 0;
while (conta < lung) {
    scanf("%d", &dato);
    somma = somma + dato;
    conta = conta + 1;
}
printf("La somma e' %d\n", somma);
```

Esempio: Leggere 10 interi **positivi** e stamparne il massimo.

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

```
int conta, dato, massimo;
printf("Immetti 10 interi: ");
massimo = 0;
conta = 0;
while (conta < 10) {
    scanf("%d", &dato);
    if (dato > massimo)
        massimo = dato;
    conta = conta + 1;
}
printf("Il massimo e' %d\n", massimo);
```

Esercizio

Leggere 10 interi **arbitrari** e stamparne il massimo.

Istruzione **for**

- ▶ I cicli visti fino ad ora hanno queste caratteristiche comuni:
 - ▶ utilizzano una variabile di controllo
 - ▶ la guardia verifica se la variabile di controllo ha raggiunto un limite prefissato
 - ▶ ad ogni iterazione si esegue un'azione
 - ▶ al termine di ogni iterazione viene incrementato (decrementato) il valore della variabile di controllo

Esempio: Stampare i numeri **pari** da 0 a N.

```
i = 0; /* Inizializzazione della var. di controllo */
while (i <= N) { /* guardia */
    printf("%d ", i); /* Azione da ripetere */
    i=i+2; /* Incremento var. di controllo */
}
```

- ▶ L'istruzione **for** permette di gestire direttamente questi aspetti:


```
for (i = 0; i <= N; i=i+2)
    printf("%d", i);
```

Sintassi:

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

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

Semantica: l'istruzione `for` precedente è equivalente a

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

Esempio:

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

- ▶ In realtà, la sintassi del **for** è

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

dove **espr-1**, **espr-2** e **espr-3** sono delle espressioni qualsiasi (in C anche l'assegnamento è un'espressione ...).

- ▶ È buona prassi:
 - ▶ 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** può 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**

Esempio: Leggere 10 interi **positivi** e stamparne il massimo. 

```
int conta, dato, massimo;
printf("Immetti 10 interi: ");
massimo = 0;
for (conta=0; conta<10; conta=conta+1)
{
    scanf("%d", &dato);
    if (dato > massimo)
        massimo = dato;
}
printf("Il massimo e' %d\n", massimo);
```

Iterazione indeterminata

- ▶ 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: Leggere una sequenza di interi che termina con 0 e calcolarne la somma.

Input: $n_1, \dots, n_k, 0$ (con $n_i \neq 0$)

Output: $\sum_{i=1}^k n_i$

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

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
```

⇒ una iterazione viene eseguita **comunque**.

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

```
main() {  
    int lunghezza = 0; /* lunghezza della sequenza */  
    int dato; /* dato letto di volta in volta */  
    printf("Inserisci una sequenza di interi (0 fine seq.)\n");  
    do {  
        scanf("%d", &dato);  
        lunghezza=lunghezza+1;  
    } while (dato != 0);  
    printf("La sequenza e' lunga %d\n", lunghezza - 1);  
}
```

- ▶ Nota: lo 0 finale non è conteggiato (non fa parte della sequenza, fa da terminatore)

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

$$\text{MCD}(12, 8) = 4$$

$$\text{MCD}(12, 6) = 6$$

$$\text{MCD}(12, 7) = 1$$

- ▶ Sfruttando direttamente la definizione di MCD
 - ▶ osservazione: $1 \leq \text{MCD}(m,n) \leq \min(m,n)$
 \implies si provano i numeri compresi tra 1 e $\min(m,n)$
 - ▶ conviene iniziare da $\min(m,n)$ e scendere verso 1

Algoritmo: stampa MCD 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

{

if `mcd` divide sia `m` che `n`

 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 a 1)
- ▶ per verificare se si è trovato il MCD si utilizza una variabile booleana (nella guardia del ciclo)
- ▶ Implementazione in C ...

```
int m, n;           /* i due numeri letti */
int mcd;           /* il massimo comun divisore */
int trovato = 0;   /* var. booleana: inizialmente false */
if (m <= n)        /*inizializza mcd al minimo tra m e n*/
    mcd = m;
else
    mcd = n;
while (mcd > 1 && !trovato)
    if ((m % mcd == 0) && (n % mcd == 0))
        /* mcd divide entrambi */
        trovato = 1;
    else
        mcd = mcd -1;
printf("MCD di %d e %d: %d", m, n, mcd);
```

Quante volte viene eseguito il ciclo?

- ▶ caso migliore: 1 volta (quando m divide n o viceversa)
es. $MCD(500, 1000)$
- ▶ caso peggiore: $\min(m,n)$ volte (quando $MCD(m,n)=1$)
es. $MCD(500, 1001)$
- ▶ l'algoritmo si comporta *male* se m e n sono grandi e $MCD(m,n)$ è piccolo

Metodo di Euclide per il calcolo del MCD

- ▶ Permette di ridursi più velocemente a numeri più piccoli, sfruttando le seguenti proprietà:

$$\text{MCD}(x, x) = x$$

$$\text{MCD}(x, y) = \text{MCD}(x-y, y) \quad \text{se } x > y$$

$$\text{MCD}(x, y) = \text{MCD}(x, y-x) \quad \text{se } y > x$$

- ▶ I divisori comuni di m ed n , con $m > n$, sono anche divisori di $m-n$.

Es.: $\text{MCD}(12, 8) = \text{MCD}(12-8, 8) = \text{MCD}(4, 8-4) = 4$

- ▶ Come si ottiene un algoritmo?

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

Esempio:

m	n	maggiore - minore
210	63	147
147	63	84
84	63	21
21	63	42
21	42	21
21	21	

Algoritmo: di Euclide per il calcolo del MCD

```
int m,n;
scanf("%d%d", &m, &n);
while (m != n)
    if (m > n)
        m = m - n;
    else
        n = n - m;

printf("MCD: %d\n", m);
```

- ▶ Cosa succede se $m=n=0$?
⇒ il risultato è 0
- ▶ E se $m=0$ e $n \neq 0$ (o viceversa)?
⇒ si entra in **un ciclo infinito**

- ▶ Per assicurarci che l'algoritmo venga eseguito su valori corretti, possiamo inserire una verifica sui dati in ingresso, attraverso un ciclo di lettura

Proposte?

```
do {  
    printf("Immettere due interi positivi: ");  
    scanf("%d%d", &m, &n);  
    if (m <= 0 || n <= 0)  
        printf("Errore: i numeri devono essere > 0!\n");  
} while (m <= 0 || n <= 0);
```

Metodo di Euclide con i resti per il calcolo del MCD

- ▶ Cosa succede se $m \gg n$?

Esempio:

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?
- ▶ Metodo di Euclide: sia

$$m = n \cdot k + r \quad (\text{con } 0 \leq r < n)$$

$$\text{MCD}(m, n) = n \quad \text{se } r=0$$

$$\text{MCD}(m, n) = \text{MCD}(r, n) \quad \text{se } r \neq 0$$

Algoritmo di Euclide con i resti per il calcolo del MCD

leggi `m` ed `n`

while `m` ed `n` sono entrambi $\neq 0$

{ sostituisci il maggiore tra `m` ed `n` con

il resto della divisione del maggiore per il minore

}

stampa il numero tra i due che e' diverso da 0

Esercizio

Tradurre l'algoritmo in C

Cicli annidati

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

Esempio: Stampa della tavola pitagorica.

Algoritmo

```
for ogni riga tra 1 e 10
  { for ogni colonna tra 1 e 10
    stampa riga * colonna
    stampa un a capo }
```

- ▶ Traduzione in C

```
int riga, colonna;
const int Nmax = 10; /* indica il numero di righe e di
colonne */
for (riga = 1; riga <= Nmax; riga=riga+1) {
  for (colonna = 1; colonna <= Nmax; colonna=colonna+1)
    printf("%d ", riga * colonna);
  putchar('\n'); }
```

Digressione sulle costanti: la direttiva `#define`

- ▶ Nel programma precedente, `Nmax` è una costante. Tuttavia la dichiarazione

```
const int Nmax = 10;
```

causa l'allocazione di memoria (si tratta duna dichiarazione di variabile *read only*)

- ▶ C'è un altro modo per ottenere un **identificatore costante**, che utilizza la direttiva **`#define`**.

```
#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
- ▶ a differenza di **`const`** **non** alloca memoria

Assegnamento e altri operatori

- ▶ In C, l'operazione di **assegnamento** $x = \text{exp}$ è un'espressione
 - ▶ il valore dell'espressione è il valore di exp (che è a sua volta un'espressione)
 - ▶ la valutazione dell'espressione $x = \text{exp}$ ha un **side-effect**: quello di assegnare alla variabile x il valore di exp
- ▶ Dunque in realtà, "=" è un operatore (associativo a destra).
Esempio: Qual'è l'effetto di $x = y = 4$?
 - ▶ È equivalente a: $x = (y = 4)$
 - ▶ $y = 4$... espressione di valore 4 con modifica (side-effect) di y
 - ▶ $x = (y = 4)$... espressione di valore 4 con ulteriore modifica su x
- ▶ L'eccessivo uso di assegnamenti come espressioni rende il codice difficile da comprendere e quindi correggere/modificare.

Operatori di incremento e decremento

▶ Assegnamenti del tipo: $i = i + 1$
 $i = i - 1$ sono molto comuni.

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

▶ In realtà ++ corrisponde a due operatori:

▶ **postincremento**: $i++$

- ▶ il valore dell'espressione è il valore di i
- ▶ side-effect: incrementa i di 1

▶ L'effetto di

```
int i, j;
```

```
i=6;
```

```
j=i++;
```

è $j=6, i=7$.

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

```
int i,j;
```

```
i=6;
```

```
j=++i;
```

è `j=7, i=7`.

(analogamente per `i--` e `--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 in questo modo!**

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

- ▶ Riscrivere, ad esempio, come segue:

4b: $z = (x++) + y;$ \implies $z = x + y;$
 $x++;$

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

Ordine di valutazione degli operandi

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

Esempio: `int x, y, z;`

`x = 2;`

`y = 4;`

`z = x++ + (x * y);`

- ▶ Quale è il valore di `z`?

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

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

Forme abbreviate dell'assegnamento

`a = a + b;` \implies `a += b;`

`a = a - b;` \implies `a -= b;`

`a = a * b;` \implies `a *= b;`

`a = a / b;` \implies `a /= b;`

`a = a % b;` \implies `a %= b;`