Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

Le strutture di controllo

Catia Prandi - catia.prandi2@unibo.it

Credit: **Pietro Di Lena**

```c
// Programming insomnia
int sheep = 0;
while(!sleep(1)) {
  sheep++;
  printf("%d\n",sheep);
}
```

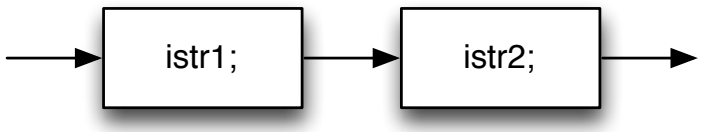
Le strutture di controllo

- Le **strutture di controllo** sono delle particolari istruzioni, tipiche dei linguaggi imperativi, che permettono di eseguire delle istruzioni secondo determinate condizioni.
- Teorema di Böhm-Jacopini (1966). Ogni algoritmo (*funzione calcolabile*) può essere implementato utilizzando tre sole *strutture di controllo*:
  1. **Sequenziali**. Blocchi di codice le cui istruzioni vengono eseguite nella stessa sequenza in cui sono organizzate nel codice sorgente. I blocchi di codice possono contenere strutture condizionali e/o iterative.
  2. **Condizionali**. Permettono di specificare due rami (o blocchi) di codice, di cui solo uno verrà eseguito in base al risultato booleano di una specifica espressione condizionale (vero/falso).
  3. **Iterative**. Permettono di eseguire iterativamente lo stesso blocco di codice, fintanto che una specifica espressione condizionale è vera.
- **Programmazione strutturata**: paradigma di programmazione che prevede la scomposizione del problema in blocchi di istruzioni da eseguire consecutivamente.
  - Il concetto di *programmazione strutturata* emerge negli anni sessanta come proposta per regolamentare e standardizzare le metodologie di programmazione, allo scopo di migliorare la qualità e la chiarezza del codice.
- Il linguaggio C mette a disposizione diversi costrutti per implementare le strutture di controllo *sequenziali*, *condizionali* e *iterative* (costrutti tipici della programmazione strutturata). Include, inoltre, costrutti per implementare *istruzioni di salto* (costrutti generalmente deprecati nella programmazione strutturata).

Blocchi di istruzioni

- Sequenza di istruzioni racchiuse tra parentesi graffe (**blocco di istruzioni**).

```
{
  <istr1>;
  <istr2>;
  ..
}
```



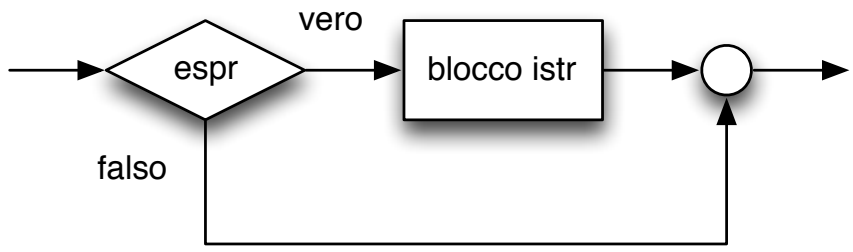
- Ogni istruzione è terminata da punto e virgola. Un ; singolo indica *istruzione vuota*.
- Le istruzioni in sequenza vengono eseguite una di seguito all'altra: non è possibile eseguire due istruzioni contemporaneamente.
- All'interno di un blocco è possibile inserire istruzioni o *annidare* altri blocchi.

```
1  #include<stdio.h>     // Direttiva, non istruzione
2
3  int main() {          // Inizio blocco main()
4    int x;              // Istruzione di dichiarazione
5    x = 0;              // Istruzione di assegnamento
6
7    {                   // Inizio blocco annidato
8      int y=x+10;       // Istruzione composta
9      ;                 // Istruzione vuota
10   }                   // Fine blocco annidato
11   printf("%d\n",x);   // Istruzione di chiamata a funzione
12   return 0;           // Istruzione return
13 }                     // Fine blocco main()
```

Il costrutto condizionale if

- Il costrutto `if` realizza un'istruzione di **salto condizionale**.

```
if (<espr>) {
  <blocco istr>
}
```



- Se l'espressione `espr` è **vera** (i.e. $\neq 0$) allora si esegue il blocco di istruzioni definito in `blocco istr`, altrimenti si prosegue con l'elaborazione alla riga successiva.
- Come per tutte le altre strutture di controllo, `blocco istr` può rappresentare una singola istruzione oppure istruzioni multiple. In quest'ultimo caso è necessario racchiudere le istruzioni tra parentesi graffe.

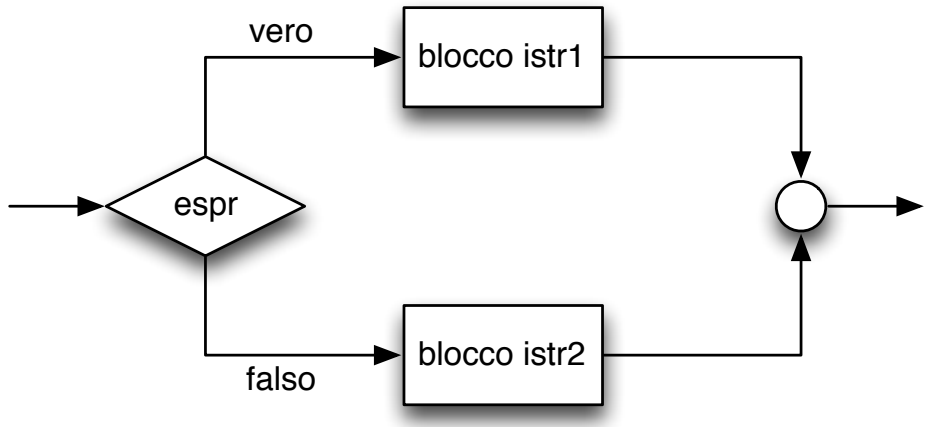
```
1  int x;
2
3  printf("Inserire un numero: ");
4  scanf("%d",&x);
5
6  if (x%2==0)
7    printf("%d: pari\n",x);
```

```
1  int x;
2
3  printf("Inserire un numero: ");
4  scanf("%d",&x);
5
6  if (x<0) {
7    x = -x;
8    printf("Numero negativo\n");
9  }
10 printf("Valore assoluto: %d: \n",x);
```

Il costrutto condizionale `if-else`

- Il costrutto `if` ammette l'enunciato *opzionale* `else`.

```
if (<espr>) {
  <blocco istr1>
} else {
  <blocco istr2>
}
```



- Le istruzioni in `blocco istr1` sono eseguite solo se l'espressione in `espr` è **vera** (i.e $\neq 0$). In caso contrario, sono eseguite le istruzioni in `blocco istr2`.

```
1 int x;
2
3 printf("Inserire un numero: ");
4 scanf("%d",&x);
5
6 if (x%2==0)
7   printf("%d: pari\n",x);
8 else
9   printf("%d: dispari\n",x);
```

```
1  int x;
2
3  printf("Inserire un numero: ");
4  scanf("%d",&x);
5
6  if (x<0) {
7    x = -x;
8    printf("Numero negativo\n");
9  } else
10   printf("Numero positivo\n");
11 printf("Valore assoluto: %d: \n",x);
```

Esempio: indovina numero (implementazione naïf)

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

#define MIN 1
#define MAX 6
#define READNUM(x) (fpurge(stdin), scanf("%d",&x)) // (fflush(stdin), scanf("%d",&x)) su WIN

int main() {
  int x, num, i=1;
  int maxt=(int)floor(log2(MAX-MIN+1))+1; // Calcola il num minimo di tentativi

  srand(time(0)); // Inizializza il generatore di numeri casuali
  num = (rand()%(MAX-MIN+1))+MIN; // Sceglie un numero in [MIN..MAX]

  printf("Indovina un numero in [%d..%d] in %d tentativi\n\n",MIN,MAX,maxt);
  printf("Quale numero scegli? ");READNUM(x);

  if(x==num) printf("Tentativo %d: hai indovinato il numero %d!\n",i++,num);
  else {
    if(x>num) printf("Tentativo %d: il numero inserito e' troppo grande\n",i++);
    else      printf("Tentativo %d: il numero inserito e' troppo piccolo\n",i++);
    printf("Quale numero scegli? "); READNUM(x);
    if(x==num) printf("Tentativo %d: hai indovinato il numero %d!\n",i++,num);
    else {
      if(x>num) printf("Tentativo %d: il numero inserito e' troppo grande\n",i++);
      else      printf("Tentativo %d: il numero inserito e' troppo piccolo\n",i++);
      printf("Quale numero scegli? "); READNUM(x);
      if(x==num) printf("Tentativo %d: hai indovinato il numero %d!\n",i++,num);
      else       printf("Tentativo %d: non hai indovinato il numero %d!\n",i++,num);
    }
  }
  return 0;
}
```

Esempio: commenti al codice di indovina numero

- Il programma sceglie un numero casuale nell'intervallo [0..5] e chiede all'utente di indovinarlo, indicando solo se il numero scelto è maggiore o minore del numero da indovinare. Abbiamo 3 tentativi a disposizione.
- Alcune considerazioni generali:
  - I valori 0 e 5 sono *hardcoded* ma definiti come macro MIN e MAX (possiamo modificarli).
  - Il numero casuale viene scelto utilizzando la funzione di libreria `rand()`, che genera un numero random nell'intervallo [0,RAND_MAX]. Per ottenere un numero random nell'intervallo [MIN..MAX] applichiamo la formula (riga 13):
    `(rand() % (MAX-MIN+1)) + MIN`.
  - La funzione di libreria `srand()` (riga 12) setta il *seed* del generatore di numeri casuali. Il seed è settato all'orario corrente (`time(0)`).
  - E' necessario includere `stdlib.h` (riga 2) per poter usare `rand()` e `srand()`. E' necessario includere `time.h` (riga 3) per poter usare `time()`.
  - La macro `READNUM(x)` (riga 7) fa uso dell'operatore virgola per liberare il buffer di memoria della `scanf()` prima della chiamata.
- Considerazioni sulla procedura:
  - E' possibile indovinare un numero casuale nell'intervallo [MIN..MAX] in al massimo $\lfloor \log_2(\text{MAX-MIN} + 1) \rfloor + 1$ passi, utilizzando un *approccio binario*. Possiamo quindi indovinare il numero in massimo $\lfloor \log_2(5 - 0 + 1) \rfloor + 1 = 3$ passi.
  - Il codice presenta esattamente 3 livelli di annidamento di costrutti `if-else`. Tutti i livelli sono ripetitivi, tranne l'ultimo.
  - Se modifichiamo l'intervallo [MIN..MAX], tanto da cambiare il valore della formula $\lfloor \log_2(\text{MAX-MIN} + 1) \rfloor + 1$, siamo costretti a modificare anche la cascata di `if-else`, rimuovendo o aggiungendo ramificazioni. Con il solo costrutto `if-else`, non possiamo fare di meglio.

Il costrutto condizionale `else-if`

- Le istruzioni `if-else` possono essere combinate in **scelta multipla**.

```
if (<espr1>) {
  <blocco istr1>
} else if (<espr2>) {
  <blocco istr2>
...
} else { <blocco istrN> }
```

- Le espressioni sono valutate nell'ordine in cui si presentano. Se una di esse è vera, il blocco di istruzioni ad essa associato viene eseguito e la catena termina.
- Ad ogni costrutto `else` deve corrispondere uno e un solo costrutto `if`.
- Regola: un costrutto `else` viene sempre associato al più vicino `if`.

```
1  char c;
2  printf("Inserire una lettera  A-D: "); scanf("%c",&c);
3  if(c=='A')
4    printf("Grade A is Excellent\n");
5  else if(c=='B')
6    printf("Grade B is Good\n");
7  else if(c=='C')
8    printf("Grade C is Satisfactory\n");
9  else if(c=='D')
10   printf("Grade D is Poor\n");
11 else
12   printf("Grade %c is Unknown\n",c);
```

Il costrutto condizionale: alcuni errori comuni

- Attenzione a dove viene posizionato il punto e virgola.

```
1 int x;
2 scanf("%d",&x);
3 if(x<0);
4   x=-x;
```

L'istruzione `x=-x` verrà sempre eseguita: è presente un `;` subito dopo l'`if` (riga 3).

- Attenzione a non farsi confondere dalle indentazioni.

```
1 int x;
2 scanf("%d",&x);
3 if(x<0)
4   x=-x;
5   printf("Warning: immesso valore negativo\n");
```

L'istruzione `printf()` verrà sempre eseguita: non appartiene al corpo dell'*if*.

- Attenzione a come vengono associati i vari rami else (nel dubbio, usare le parentesi).

```
1 int x=-10;
2 if (x<1)
3   if (x>-1) printf("x uguale a 0\n");
4 else printf("Errore: x>=1\n");
```

Viene stampato `"Errore: x>=1"`: l'`else` è associato all'`if` più vicino.

Il costrutto di selezione switch

- Il costrutto `switch` realizza un'istruzione di scelta multipla. Sintassi generale

```
switch (<int espr>) {
    case <espr-const1> :  <blocco istr1> [break;]
    case <espr-const2> :  <blocco istr2> [break;]
    ...
    default:  <blocco istr> [break;]
}
```

- Se il valore di `int espr` coincide con il valore specificato in uno dei vari `case`, il blocco di istruzioni corrispondente viene eseguito.
- L'istruzione `default` (opzionale) viene eseguita se gli altri `case` non sono soddisfatti.
- Le `espr-const` devono essere **costanti intere**, `int espr` deve essere di tipo intero.
- L'istruzione `break` (opzionale) provoca l'uscita dallo `switch`. Se non utilizzato, viene eseguito il blocco di istruzioni del `case` successivo.

```
1 char c;
2 printf("Inserire una lettera  A-D: "); scanf("%c",&c);
3 switch(c) {
4   case 'A': printf("Grade A is Excellent\n");    break;
5   case 'B': printf("Grade B is Good\n");         break;
6   case 'C': printf("Grade C is Satisfactory\n");break;
7   case 'D': printf("Grade D is Poor\n");         break;
8   default:  printf("Grade %c is Unknown\n",c);
9 }
```

Il costrutto di selezione switch: esempi corretti

- Le etichette possono essere espressioni costanti di qualsiasi tipo (macro, caratteri, ecc)

```
1  #define N 7
2  int x;
3  printf("Inserire una cifra intera 0-9: ");
4  scanf("%d",&x);
5
6  switch(x) {
7    case 2:       // 2
8    case 1+2:     // 3
9    case 'A'-60:  // 5
10   case N:       // 7
11       printf("%d: numero primo\n",x);
12 }
```

- Le etichette e il costrutto default possono essere in qualsiasi ordine.

```
1  int x;
2  printf("Inserire una cifra intera 0-9: ");
3  scanf("%d",&x);
4
5  switch(x) {
6    default:  printf("Ammessi solo numeri a singola cifra\n"); break;
7    case 0: case 2: case 4: case 6: case 8:
8       printf("%d: pari\n",x); break;
9    case 1: case 3: case 5: case 7:  case 9:
10      printf("%d: dispari\n",x); break;
11  }
```

Il costrutto di selezione `switch`: esempi non corretti

- Alcuni esempi di etichette non valide.

```c
int x=0, y=0;

switch(x) {
  case 2.1:    // Errore: tipo float
  case "abc": // Errore: stringa costante
  case 'a':    // OK
  case 'a':    // Errore: etichetta duplicata
  case y:      // Errore: variabile
  case >0:     // Errore: operatore relazionale
  default: printf("Done\n");
}
```

- Esempio sintatticamente corretto ma con comportamento non "pulito".

```c
switch(x) {
  printf("Before case\n"); // Non viene mai eseguita
  case 0:  printf("In case 0\n"); // Manca il break;
  case 1:  printf("In case 1\n");     break;
  default: printf("Default case\n"); break;
  printf("After case\n"); // Non viene mai eseguita
}
```

Le strutture di controllo iterative

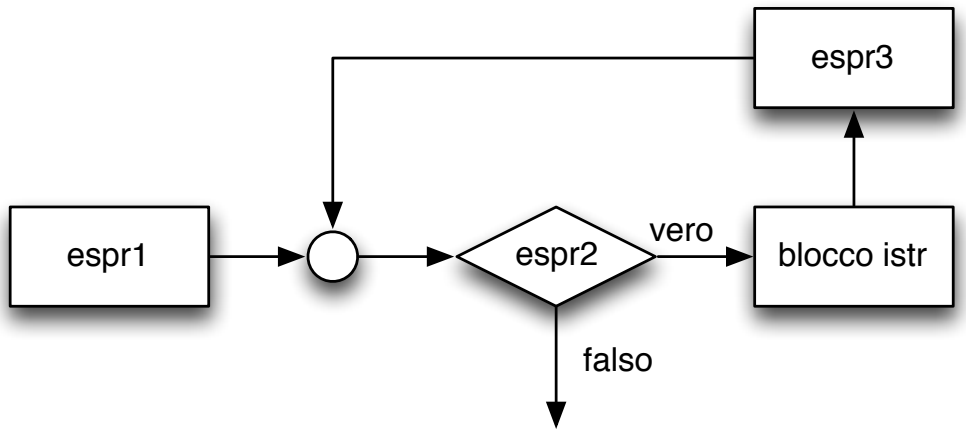
- Le strutture di controllo **iterative** permettono di specificare che un dato blocco di istruzioni sia eseguito **ripetutamente**, fintanto che determinate **condizioni di permanenza nel ciclo** siano soddisfatte.
- Per il teorema di Jacopini-Böhm, un linguaggio che ammette unicamente strutture sequenziali e condizionali ha un potere espressivo limitato.
- Esempio: scrivere un algoritmo che calcoli la la potenza tra due interi, utilizzando solo il costrutto `if-else`. Soluzione ingenua:

```c
#include<stdio.h>

int main() {
 int n,m;
 printf("Inserisci una potenza tra interi [n^m]: ");
 scanf("%d^%d",&n,&m);
      if (m==0)  printf("%d^%d = %d\n",n,m,1);
 else if (m==1)  printf("%d^%d = %d\n",n,m,n);
 else if (m==-1) printf("%d^%d = %g\n",n,m,1.0/n);
 else if (m==2)  printf("%d^%d = %d\n",n,m,n*n);
 else if (m==-2) printf("%d^%d = %g\n",n,m,1.0/(n*n));
 else if (m==3)  printf("%d^%d = %d\n",n,m,n*n*n);
 else if (m==-3) printf("%d^%d = %g\n",n,m,1.0/(n*n*n));
 else printf("Mi arrendo!\n");

 return 0;
}
```

Il costrutto iterativo `for`

- Il costrutto `for` serve per realizzare un **ciclo** (o **loop**), cioè l'esecuzione dello stesso insieme di istruzioni per un certo numero di volte.

```
for (<espr1>; <espr2>; <espr3>) {
  <blocco istr>
}
```



- **Inizializzazione**. Prima di iniziare il ciclo, viene valutata una sola volta `espr1`.
- **Test**. L'espressione `espr2` è una condizione di permanenza nel ciclo: le istruzioni in `blocco istr` sono eseguite solo se `espr2` è vera. Diversamente, il ciclo termina.
- **Incremento**. L'espressione `espr3` viene valutata al termine di ogni iterazione, i.e. dopo aver eseguito le istruzioni in `blocco istr`.
- Il valore di `espr1` e `espr3` è ignorato (sono valutate come espressioni `void`).
  - Possono, ad esempio, essere chiamate a funzione che non ritornano un valore.

Il costrutto iterativo `for`: considerazioni generali

- Nei linguaggi di programmazione, il costrutto `for` è generalmente impiegato per eseguire una serie di istruzioni per un cento numero **noto** di volte.

```c
// Stampa i numeri da 1 a n
int i, n = 100;
for(i = 1; i <= n; i++)
  printf("%d\n",i);
```

```c
// Stampa i numeri  da n a 1
int i, n = 100;
for(i = n; i >= 1; i--)
  printf("%d\n",i);
```

  - In questi esempi, la variabile `i` è utilizzata come **contatore**: è inizializzata ad un **valore iniziale** e poi incrementata o decrementata ad ogni iterazione fino al raggiungimento di un **valore finale** `n`.
  - La variabile `i` ha il ruolo di **contare** il numero di volte che si esegue il ciclo.
  - Possiamo leggere il codice come: "*stampa il valore di* i*, per* i *che va da 1 a* n".
- Nel linguaggio C (e i sui derivati), il costrutto `for` assume una connotazione più generale: permette di definire cicli con condizioni di terminazione *complesse*.

```c
// Legge e stampa i singoli caratteri di
// una singola sequenza terminata da '\n'.
int i;
char c;
for(i=1; (c=getchar())!='\n'; i++)
  printf("Carattere %d: %c\n",i,c);
```

In questo esempio, la condizione di terminazione è la lettura del carattere `'\n'`.

Esempio: una soluzione al calcolo di potenze tra interi

```
1  #include<stdio.h>
2
3  int main() {
4   int n,m,i;
5   long int pow = 1;
6
7   printf("Inserisci una potenza tra interi [n^m]: ");
8   scanf("%d^%d",&n,&m);
9
10  for(i = (m >= 0 ? m : -m); i > 0; i--)
11      pow *= n;
12  if(m>=0)  printf("%d^%d = %ld\n",n,m,pow);
13  else      printf("%d^%d = %g\n",n,m,1.0/pow);
14
15  return 0;
16 }
```

- Calcola la potenza tra due numeri interi, utilizzando il costrutto iterativo `for`.
- A livello teorico, l'algoritmo funziona correttamente per qualsiasi coppia di interi n e m. A livello pratico, funziona correttamente solo su un limitato insieme di coppie (n,m). Aumentare la dimensione dei tipi di dato delle variabili di calcolo non risolve il problema: per quasi tutte le coppie (n,m) la variabile `pow` andrà in overflow.
- Il codice, non considera/gestisce le situazioni di overflow.
- Non è l'algoritmo più efficiente in termini di tempo di calcolo. Possiamo fare di meglio.

Esempio: calcolo del fattoriale

```c
#include <stdio.h>

// Calcola il fattoriale di un intero positivo n.

int main() {
 unsigned int n;
 unsigned long int i, res = 1;

  printf("Inserici un intero positivo: ");
  scanf("%u",&n);

  for(i = 2; i <= n; i++)
    res *= i;

  printf("%d! = %lu\n",n,res);

  return 0;
}
```

- Programma che calcola il fattoriale $n!$ di un numero n.
- Definizione: $n! = n * (n-1) * (n-2) * ... * 2 * 1$.
- Per convenzione: $0! = 1$.

Esempio: stampa della codifica binaria

```
1  #include<stdio.h>
2  #include<limits.h>
3
4  // Stampa la codifica binaria di un int
5  int main() {
6   int x,i;
7
8   printf("Inserisci un intero in [%d,%d]: ",INT_MIN,INT_MAX);
9   scanf("%d",&x);
10
11  for(i = 8*sizeof(int)-1; i >= 0; i--)
12    x & (1<<i) ? putchar('1') : putchar('0');
13
14  putchar('\n');
15  return 0;
16 }
```

- Utilizzando il costrutto `for` possiamo facilmente generalizzare l'algoritmo di stampa della codifica binaria per tipi differenti.
- Per gestire un tipo di dato intero diverso da `int` è sufficiente cambiare poche righe di codice.
- "x « y" Calcola la shift a sinistra del primo operando per un numero di bit pari al valore del secondo operando.

Il costrutto iterativo `for`: esempi 1/3

- Il costrutto `for` è estremamente *versatile*: le espressioni di *inizializzazione*, *test* e *incremento* sono opzionali. Perché è comunque necessario introdurre i `;`?

```c
// Stampa i numeri
// da 1 a n
int i=1, n=100;
for(; i<=n; i++)
  printf("%d\n",i);
```

```c
// Stampa i numeri
// da 1 a n
int i=1, n=100;
for(i=1; i<=n; )
  printf("%d\n",i++);
```

```c
// Stampa i numeri
// da 1 a n
int i=1, n=100;
for(;i<=n;)
  printf("%d\n",i++);
```

Gli esempi sopra sono tutti semanticamente equivalenti.

- E' possibile definire *cicli infiniti*, che non terminano mai. A meno che questo comportamento non sia esplicitamente cercato, i seguenti sono esempi di codice da evitare.

```c
// Ciclo infinito
int i=1, n=100;
for(i=1; i>= -n ; i++)
  printf("%d\n",i);
```

```c
// Ciclo infinito
int i=1;
for(;1;)
  printf("%d\n",i++);
```

```c
// Ciclo infinito
int i=1;
for(;;)
  printf("%d\n",i++);
```

Nessuno dei tre esempi sopra termina il calcolo, in quanto l'espressione di test è sempre vera.

Il costrutto iterativo `for`: esempi 2/3

- Possiamo utilizzare più espressioni di controllo, facendo uso dell'operatore virgola.

```
1 // Stampa tutte le coppie i+j=100
2 int i, j, n=100;
3 for(i=0, j=n; i<=n && j>=0; i++, j--)
4   printf("%d + %d = %d\n",i,j,i+j);
```

Il ciclo continua fintanto che `i<=n` e allo stesso tempo `j>=0`. Letta al contrario, il ciclo termina quando `i > n` oppure `j < 0`.

- I cicli for possono essere annidati ad un qualunque livello di profondità

```
1 // Stampa la tavola pitagorica
2 int i,j;
3
4 for(i=1; i<=10; i++) {
5   for(j=1; j<=10; j++)
6     printf("%3d ",i*j);
7   printf("\n");
8 }
```

Il ciclo più interno (indice `j`) viene *riattivato* ad ogni iterazione del ciclo più esterno (indice `i`).

Il costrutto iterativo `for`: esempi 3/3

- Quali dei due esempi è maggiormente *efficiente* in termini di tempo di calcolo?

```c
// Stampa i numeri da 0 a n^2
int i, n;
scanf("%d",&n);

for(i=0; i<=n*n; i++)
  printf("%d\n",i);
```

```c
// Stampa i numeri da 0 a n^2
int i, n, k;
scanf("%d",&n);

for(i=0, k=n*n; i<=k; i++)
  printf("%d\n",i);
```

Nel secondo esempio, il valore di controllo `n*n`, che serve a determinare la condizione di arresto del ciclo, viene calcolato una sola volta. Nel primo esempio, è necessario ri-calcolarlo ogni volta che viene valutata l'espressione di test, sprecando risorse di calcolo.

- Quali dei due esempi è maggiormente *efficiente* in termini di tempo di calcolo?

```c
// Stampa i numeri da n a 0
unsigned int i, n;
scanf("%u",&n);

for(i=n; i>=0; i--)
  printf("%u\n",i);
```

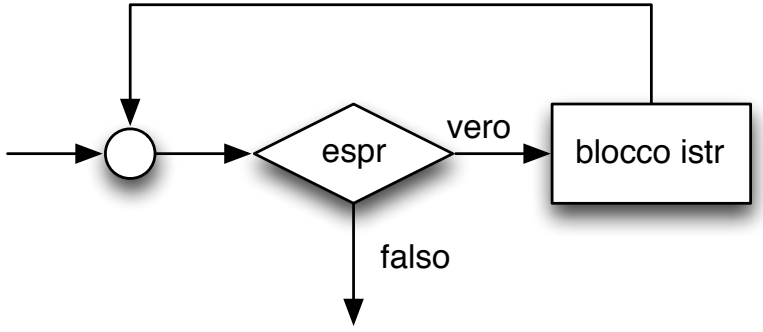
```c
// Stampa i numeri da n a 0
unsigned int i, n;
scanf("%u",&n);

for(i=n+1; i--; )
  printf("%u\n",i);
```

Il secondo esempio è più veloce ma anche meno leggibile. Morale: trovare un compromesso tra leggibiltà ed efficienza del codice.

Il costrutto iterativo `while`

- Il costrutto `while`, come il costrutto `for`, serve per realizzare un **ciclo**.

```
while (<espr>) {
  <blocco istr>
}
```



- Le istruzioni in `blocco istr` sono eseguite fintanto che `espr` è vera.
- L'espressione `espr` viene valutata prima di eseguire le istruzioni nel blocco. Il `blocco istr` potrebbe quindi non essere mai eseguito.

```
1 int i=1;
2 char c;
3
4 while((c=getchar())!='\n')
5   printf("Carattere %d: %c\n",i,c);
```

```
1 int i=1, n=100;
2
3 while(i<=n)
4   printf("%d\n",i++);
```

Nel primo esempio, il corpo del `while` non verrà eseguito se da tastiera si preme il tasto di *a capo* (invio).

Il costrutto iterativo `while`: esempi

- Calcola la somma di una serie di interi. Si ferma quando viene inserito il numero 0.

```
1  int x, sum=0;
2
3  printf("Inserisci un intero [0 per terminare]: ");
4  scanf("%d",&x);
5  while(x!=0) {
6    sum+=x;
7    printf("Inserisci un intero [0 per terminare]: ");
8    scanf("%d",&x);
9  }
10 printf("Somma: %d\n",sum);
```

- Esempi di ciclo infinito con il `while`.

```
1  int i=0;
2
3  while(1)
4    printf("%d\n",i++);
```

```
1  int i=0, n;
2
3  scanf("%d",&n);
4  while(i>n)
5    printf("%d\n",i++);
```

Nel primo caso la condizione di terminazione è sempre vera (1). Nel secondo caso, il costrutto `while` andrà in loop se l'utente immette un numero minore di 0, non verrà invece eseguita nessuna `printf()` se il numero immesso è maggiore o uguale di 0.

Esempio: la sequenza di Collatz

- La **congettura di Collatz** è un problema matematico, tuttora irrisolto.
- Enunciata per la prima volta da Lothar Collatz nel 1937.
- Consideriamo la seguente procedura:
  1. Prendiamo un intero positivo n.
  2. Se $n = 1$ l'algoritmo termina.
  3. Se n è pari lo si divide per due, altrimenti lo moltiplichiamo per 3 e sommiamo 1.

$$f(n) = \begin{cases} n/2 & \text{se n è pari} \\ 3n+1 & \text{se n è dispari} \end{cases}$$

- Possiamo generare una sequenza di valori, applicando la funzione $f(n)$ ad un qualsiasi intero positivo. Ad esempio, partendo da $n = 5$:

$$f(5) = 16 \rightarrow f(16) = 8 \rightarrow f(8) = 4 \rightarrow f(4) = 2 \rightarrow f(2) = 1 \Rightarrow \text{stop!}$$

- La congettura di Collatz asserisce che questa sequenze giungerà sempre al termine, indipendentemente dal numero di partenza.
- E' stata verificata computazionalmente per valori fino a 5×2^{60}, senza trovare un controesempio.

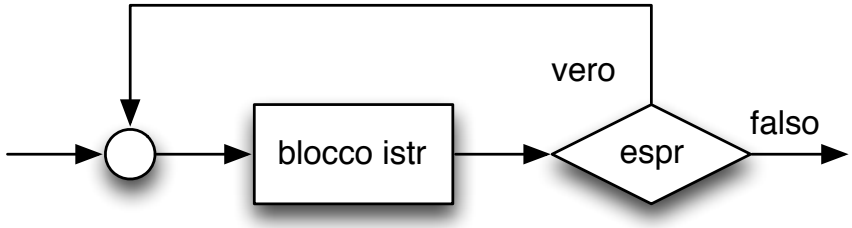
Esempio: algoritmo per stampare la sequenza di Collatz

```c
#include <stdio.h>

#define UNIX

#ifdef UNIX
#define READNUM(x) (fpurge(stdin), scanf("%u",&x))
#else
#define READNUM(x) (fflush(stdin), scanf("%u",&x))
#endif

int main() {
  unsigned int x = 0, i = 1;

  while(x < 1) {
    printf("Inserici un intero >= 1: ");
    READNUM(x);
  }
  printf("Numero di partenza: %d\n",x);
  while(x != 1) {
    if(x % 2) x  = 3*x+1;
    else      x /= 2;
    printf("Passo %d: %d\n",i++,x);
  }
  return 0;
}
```

Il costrutto iterativo do-while

- Il costrutto `do-while`, come i costrutti `for` e `while`, serve per realizzare un **ciclo**.

```
do {
   <blocco istr>
} while (espr);
```



- A differenza del costrutto `while`, il blocco di istruzioni `blocco istr` viene eseguito sempre almeno una volta.
- L'espressione `espr`, che controlla le condizioni di terminazione, viene verificata al termine del ciclo.
- Notare il `;` dopo il `while`.
- La sintassi del `do-while` è estremamente compatta ed indicata per alcuni cicli specifici, come ad esempio il controllo di valori in input.

```
1 // Richiede iterativamente un numero in input. Termina
2 // non appena legge un intero  nell'intervallo [-2,2].
3 int x;
4 do {
5   printf("Inserisci un numero compreso tra -2 e 2: ");
6   scanf("%d",&x);
7 } while(x<-2 || x>2);
```

Il costrutto iterativo do-while: esempi

- Calcola la somma di una serie di interi. Si ferma quando viene inserito il numero 0. Confrontare con lo stesso codice scritto utilizzando il costrutto `while`.

```c
int x, sum=0;

do {
  printf("Inserisci un intero [0 per terminare]: ");
  scanf("%d",&x);
  sum += x;
} while(x!=0);
printf("Somma: %d\n",sum);
```

- Esempi di ciclo infinito con il `do-while`. Confrontare con lo stesso codice scritto utilizzando il costrutto `while`.

```c
int i=0;

do
  printf("%d\n",i++);
while(1);
```

```c
int i=0, n;

scanf("%d",&n);
do
  printf("%d\n",i++);
while(i>n);
```

Nel primo caso la condizione di terminazione è sempre vera (1). Nel secondo caso, il costrutto `do-while` andrà in loop se l'utente immette un numero minore o uguale a 0, verrà stampato soltanto il valore 0, in caso contrario.

Esempio: indovina numero (implementazione generalizzata)

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

#define UNIX
#define MIN 1
#define MAX 100

#ifdef UNIX
#define READNUM(x) (fpurge(stdin), scanf("%d",&x))
#else
#define READNUM(x) (fflush(stdin), scanf("%d",&x))
#endif

int main() {
  int x, num, i=1;
  int maxt=(int)floor(log2(MAX-MIN+1))+1; // Calcola il num minimo di tentativi
  int stop = 0;

  srand(time(0)); // Inizializza il generatore di numeri casuali
  num = (rand()%(MAX-MIN+1))+MIN; // Sceglie un numero in [MIN..MAX]
  printf("Indovina un numero in [%d..%d] in %d tentativi\n\n",MIN,MAX,maxt);

 do {
    printf("Quale numero scegli? ");READNUM(x);

    if((stop = x==num))
      printf("Tentativo %d: hai indovinato il numero %d!\n",i++,num);
    else if((stop = i==maxt))
      printf("Tentativo %d: non hai indovinato il numero %d!\n",i,num);
    else if(x>num)
      printf("Tentativo %d: il numero inserito e' troppo grande\n",i++);
    else
      printf("Tentativo %d: il numero inserito e' troppo piccolo\n",i++);
  } while(!stop);
  return 0;
}
```

Esempio: commenti al codice di indovina numero generalizzato

- Facendo uso di un costrutto iterativo possiamo facilmente generalizzare il programma "indovina numero" a intervalli generici (o meglio, ad un intervallo di interi rappresentabile con il tipo di dato `int`).
- Per modificare l'intervallo di numeri è sufficiente cambiare i valori delle macro `MIN` e `MAX` (riga 7 e 8).
- Possiamo utilizzare anche intervalli negativi.
- Il programma dà all'utente esattamente il numero minimo di tentativi sufficienti ad indovinare il numero (riga 18).
- Le condizioni di terminazione sono controllate dalla variabile `stop`, che diventa `1` solo se il numero è stato indovinato (riga 28) oppure se si è raggiunto il numero massimo di tentativi (riga 30).

Equivalenze tra i costrutti iterativi

- Il linguaggio C introduce diversi costrutti iterativi, nonostante il Teorema di Böhm-Jacopini affermi che sia sufficiente un solo costrutto iterativo per dotare il linguaggio della massima espressività.
- La scelta di dotare il linguaggio di diversi costrutti iterativi è motivata dalla decisione di fornire al programmatore la possibilità di scegliere le strutture di iterazione sintatticamente più adatte alle istruzioni da codificare (codice più leggibile e più corto).
- Utilizzando terminologia inglese, potremmo definire le varie alternative per le strutture di controllo come *syntactic sugar* (zucchero sintattico).
- I tre costrutti di iterazione possono essere facilmente e intuitivamente convertiti uno nell'altro.

Equivalenze tra costrutti iterativi: esempio 1/2

- Da `for` in `while`. Nota: non funziona in presenza di `continue` in `istruzioni`.

```
for (espr1; espr2; espr3) {
  istruzioni;
}
```

$\Longrightarrow$

```
espr1;
while(espr2) {
  istruzioni;
  espr3;
}
```

- Da `for` in `do-while`. Nota: come sopra.

```
for (espr1; espr2; espr3) {
  istruzioni;
}
```

$\Longrightarrow$

```
espr1;
if (espr2) {
  do {
    istruzioni;
    espr3;
  } while (espr2);
}
```

- Da `while` in `do-while`.

```
while(espr) {
  istruzioni;
}
```

$\Longrightarrow$

```
if (espr) {
  do {
    istruzioni;
  } while (espr);
}
```

Equivalenze tra costrutti iterativi: esempio 2/2

- Da `while` in `for`.

```
while(espr) {
  istruzioni;
}
```

$\Longrightarrow$

```
for (; espr; ) {
  istruzioni;
}
```

- Da `do-while` in `for`. Nota: potrebbe non funzionare correttamente se `istruzioni` contiene dichiarazioni di variabili.

```
do {
  istruzioni;
} while(espr);
```

$\Longrightarrow$

```
{istruzioni;
for (; espr; ) {
  istruzioni;
}}
```

- Da `do-while` in `while`. Nota: come sopra.

```
do {
  istruzioni;
} while(espr);
```

$\Longrightarrow$

```
{istruzioni;
while (espr) {
  istruzioni;
}}
```

Le istruzioni di salto

- Le istruzioni di salto permettono di *alterare* il flusso di esecuzione di una sequenza di istruzioni.
- Il linguaggio C specifica 4 istruzioni di salto:
  - `break`: forza l'**uscita** da un ciclo o dallo `switch` (già visto).
  - `continue`: forza il **riavvio** di un ciclo.
  - `goto`: forza il **salto incondizionato** ad un punto della stessa funzione (ampiamente *deprecato*).
  - `return`: termina l'esecuzione di una funzione e fa **ritornare** il controllo al chiamante. L'abbiamo vista solo nella funzione `main()`), ne parleremo quando vedremo come definire funzioni.
- L'utilizzo di istruzioni di salto nella programmazione strutturata è alquanto controverso: il loro uso/abuso è visto come sinonimo di cattivo stile di programmazione.
- Ad eccezione dell'istruzione `return` e `break` (nello `switch`) è consigliabile evitare di utilizzarle.
- Motivazione: se utilizzate in maniera impropria rendono il codice illeggibile.
- Perchè le vediamo? Per comprendere meglio quali sono le differenze tra codice considerato leggibile e *spaghetti code*.

Il costrutto di salto break

- Il costrutto `break` serve a forzare l'uscita da un ciclo o dal costrutto `switch`.
- Quando è utilizzato in cicli annidati, determina l'uscita dal loop più interno nel quale è contenuto.

```
1  char c;
2  int i=1;
3
4  do {
5    if((c = getchar()) == '\n')
6      break;
7
8    printf("Carattere %d: %c\n",i++,c);
9  } while(1);
10 // Punto di uscita
```

```
1  int i=1;
2  while(i<10) {
3    int j=1;
4    while(j<10) {
5      if(j++>i)
6        break;
7      printf("*");
8    } // Punto di uscita
9    printf("\n");
10   i++;
11 }
```

- Non può essere utilizzato con il costrutto `if`, se tale costrutto non è annidato in un ciclo o in uno `switch` (errore di compilazione).

```
1  int x=10;
2  if(x>=0) {
3    if(x==0)
4      break;   // Errore
5    else
6      printf("%d\n",x);
7  }
```

Il costrutto di salto continue

- Il costrutto `continue` può essere utilizzato solo con le strutture iterative.
- A differenza dell'istruzione `break` che causa la terminazione di un ciclo, il costrutto `continue` salta tutto il codice che la segue e prosegue con l'iterazione successiva.

```
1  // Loop infinito: legge e stampa i singoli caratteri di ogni
2  // sequenza terminata da '\n'
3  char c;
4  int i=1;
5
6  do {
7    if((c = getchar()) == '\n')
8      continue;
9    printf("Carattere %d: %c\n",i++,c);
10 } while(1);   // Punto di arrivo salto
```

- Quando usata col ciclo `for`, il salto continua con l'istruzione di incremento.

```
1  // Stampa i numeri dispari
2  // da 1 a 99
3  int i;
4  // Punto di arrivo salto: i++
5  for(i=1; i<=100; i++) {
6    if(i%2==0) continue;
7    printf("%d\n",i);
8  }
```

```
1  // Loop infinito
2  int i;
3  i=1;
4  while(i<=100) {//Punto di arrivo salto
5    if(i%2==0) continue;
6    printf("%d\n",i);
7    i++;
8  }
```

Nota: caso in cui la ¨regola¨ di conversione da ciclo `for` a `while` non funziona.

Esempio: leggibilità del codice

- L'abuso di `break` e `continue` può rendere il codice poco leggibile.

```
1 int i=0, n;
2 scanf("%d",&n);
3 while(++i<=n) {
4   if(i%3!=0) continue;
5   printf("%d\n",i);
6   if(i>48) break;
7 }
```

Se $n \geq 51$ stampa i multipli di 3 compresi tra 1 e 51, altrimenti quelli tra 1 e n.

- Il ciclo `while` ha due possibili **condizioni di terminazione** (o **punti di uscita**):
  1. Riga 6. Uscita solo se `n >= 51`. Sono stampati i multipli di 3 compresi tra 1 e 51.
  2. Riga 3. Uscita solo se `n < 51`. Sono stampati i multipli di 3 compresi tra 1 e `n`.
- Se `n >= 51`, l'istruzione `continue` (riga 4) causa il salto del `break` (riga 6) anche quando ci sarebbero le condizioni per poterlo eseguire, i.e. `i=49`, `i=50`.

- Esempio di confronto.

```
1 int i=0, n;
2 scanf("%d",&n);
3 for(i=1; i<=n && i<=51; i++)
4   if(i%3==0) printf("%d\n",i);
```

Versione semanticamente equivalente del codice precedente in cui le condizioni di uscita dal ciclo sono espresse in modo molto più leggibile.

Il costrutto di salto goto

- Il costrutto goto permette di effettuare un **salto incodizionato** dal punto in cui è specificata tale istruzione ad una *label* all'interno della stessa funzione.

```
1  // Stampa i numeri
2  // da 1 a n
3  int i, n=100;
4
5  i=1;
6  LOOP:
7  if(i<=n) {
8      printf("%d\n",i);
9      i++;
10     goto LOOP;
11 }
```

$\Longrightarrow$

```
1  // Stampa i numeri
2  // da 1 a n
3  int i, n=100;
4  for(i=1; i<=n; i++)
5    printf("%d\n",i);
```

- L'uso del goto è stato ampiamente criticato nella comunità scientifica come la principale causa dello *spaghetti code*.
- A partire dagli anni '70 si è verificato un graduale declino dell'uso di goto nel codice, anche in seguito allo spostamento verso il paradigma di programmazione strutturata.
- Per il Teorema di Böhm-Jacopini, ogni blocco di istruzioni che fa uno di goto può essere riscritto utilizzando le strutture di controllo sequenziali, condizionali e iterative.