Istruzioni

- Finora abbiamo imparato come fare ad eseguire un programma che prevede *istruzioni* semplici
  - lettura di dati di input
  - calcolo di risultati tramite espressioni
  - stampa dei risultati
- Non tutti i programmi si possono scrivere così: ci serve qualche struttura in più.
- Le *istruzioni* esprimono *azioni* che, una volta eseguite, comportano una *modifica permanente dello stato interno* del programma o del mondo circostante.
- Le *strutture di controllo* permettono di aggregare istruzioni semplici in istruzioni più complesse.

ISTRUZIONI

- Un'istruzione C è espressa dalle seguenti produzioni:

```
<istruzione> ::=  <istruzione-semplice>
<istruzione> ::=  <istruzione-di-controllo>

<istruzione-semplice> ::= <espressione> ;
```

ISTRUZIONI SEMPLICI

- Qualsiasi *espressione* seguita da un punto e virgola è una *istruzione semplice*.

- Esempi

```
x = 0; y = 1;/* due istruzioni */
x = 0, y = 1;/* una istruzione */
k++;
3;                  /* non fa nulla */
;                   /* istruz. vuota*/
```

ISTRUZIONI DI CONTROLLO

- Una istruzione di controllo può essere:
  - una istruzione *composta* (blocco)
  - una istruzione *condizionale* (selezione)
  - una istruzione di *iterazione* (ciclo)

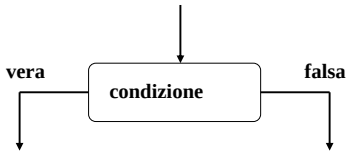
Le istruzioni di controllo sono alla base della programmazione strutturata (Dijkstra, 1969).

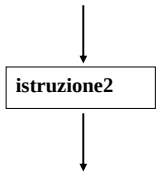
PROGRAMMAZIONE STRUTTURATA

- **Obiettivo:** rendere più facile la lettura dei programmi (e quindi la loro modifica e manutenzione).
- Abolizione di salti incondizionati (**go to**) nel flusso di controllo.
- La parte esecutiva di un programma viene vista come un comando (complesso) ottenuto da **istruzioni elementari**, mediante alcune regole di composizione (**strutture di controllo**).

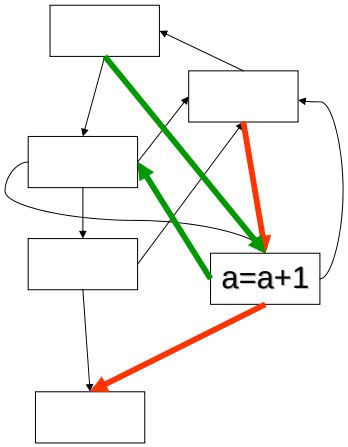
DIAGRAMMI DI FLUSSO

- Rappresentano graficamente il flusso di esecuzione di un programma
- Due componenti fondamentali



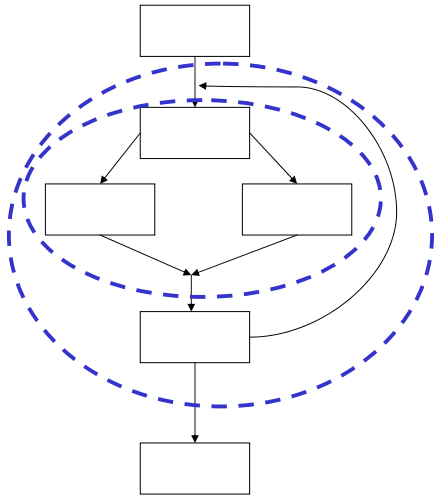


Un tempo si programmava così



- Se ho bisogno di modificare un blocco di programma, devo considerare tutti i possibili punti da cui ci si può arrivare ➔ modificare un programma diventa quasi impossibile
- Ad esempio, può succedere di accorgersi di un errore in un blocco: analizzo il programma seguendo le frecce rosse e mi accorgo che a=a+1 dovrebbe essere a=a+2
- Però io posso eseguire quell'istruzione anche seguendo le frecce verdi ed in questo caso a=a+1 è corretto.
- Come posso arrivare ad un programma corretto??

Nella programmazione strutturata



- Ogni struttura ha un solo punto di ingresso ed uno di uscita, quindi può essere considerato una "macro-istruzione"
- So quindi sempre da dove arrivo ad un certo blocco, quindi riesco a capire meglio qual è il valore delle variabili e riesco a fare le modifiche
- Non ci sono "incroci"
- Il programma è più facile da comprendere

STRUTTURE DI CONTROLLO

Concetti chiave:

- **concatenazione o composizione BLOCCO**
- **istruzione condizionale SELEZIONE**
  - ramifica il flusso di controllo in base al valore vero o falso di una espressione ("*condizione di scelta*")
- **ripetizione o iterazione CICLO**
  - esegue ripetutamente un'istruzione finché rimane vera una espressione ("*condizione di iterazione"*)

Scelte

- Voglio scrivere un programma che legge un numero in ingresso e, a seconda che il numero sia pari o dispari, visualizza la scritta "pari" o la scritta "dispari"
- Algoritmo:

  leggi x;

  calcola r = resto della divisione fra x e 2

  se r == 0

  stampa "pari"

  altrimenti stampa "dispari"
- Ho bisogno di un'istruzione che, a seconda se è verificata o meno una condizione, esegua la prima istruzione oppure la seconda

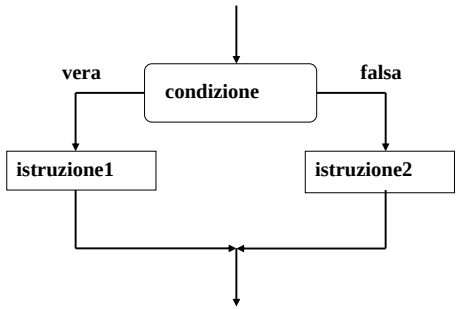
ISTRUZIONI CONDIZIONALI

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

- la seconda *non è essenziale*.
- l'espressione condizionale ternaria (.. ? … : …) fornisce *già* un mezzo per fare scelte, ma è *poco leggibile* in situazioni di medio/alta complessità. L'istruzione di scelta fornisce un altro modo per esprimere alternative.

ISTRUZIONE DI SCELTA SEMPLICE

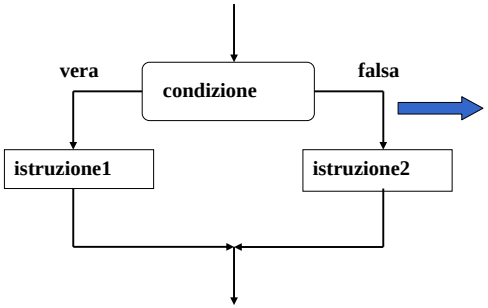
```
<scelta> ::=  if (<cond>) <istruzione1>
              [ else <istruzione2> ]
```



La condizione viene valutata al momento dell'esecuzione dell'if.

ISTRUZIONE DI SCELTA SEMPLICE

```
<scelta> ::=  if (<cond>) <istruzione1>
              [ else <istruzione2> ]
```



La parte **else** è *opzionale*: se omessa, in caso di condizione falsa si passa subito all'istruzione che segue l'**if**.

Esempio

```
#include <stdio.h>

main()
{     int x, r;
      printf("Inserisci x: ");
      scanf("%d",&x);
      r = x % 2;
      if (r==0)
            printf("pari");
            // eseguita se la condizione è vera
      else printf("dispari");
           // eseguita se la condizione è falsa
      printf("\nFine Programma\n");
      /* eseguita comunque (fuori dall'if)*/
}
```

Problema

- Es. nella divisione, voglio controllare che il divisore non sia zero.
- Algoritmo:

  leggi dividendo e divisore
  se divisore != 0
  calcola il quoziente
  calcola il resto
  visualizza quoziente e resto
  altrimenti stampa "Divisione impossibile"

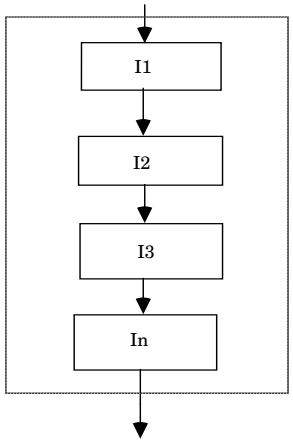
Idea: devo raggruppare una sequenza di istruzioni in un'unica istruzione

- Problema: devo eseguire più di una istruzione in uno dei due rami?

BLOCCO

```
<blocco> ::= {
  [ <dichiarazioni e
    definizioni> ]
  { <istruzione> }
}
```

- Il campo di visibilità dei simboli definiti nel blocco è ristretto al blocco stesso
- dopo un blocco non occorre il punto e virgola (esso *termina* le istruzioni semplici, non *separa* istruzioni)



ESEMPIO di BLOCCO

Abbiamo già visto un esempio di blocco: nel main c'è sempre un blocco

```
/* programma che letti due numeri a
   terminale ne stampa la somma*/
#include <stdio.h>
main()
{/* INIZIO BLOCCO */
  int X,Y;
  printf("Inserisci due numeri ");
  scanf("%d%d",&X,&Y);
  printf("%d",X+Y);
} /* FINE BLOCCO */
```

Blocco all'interno di una selezione

```
#include <stdio.h>

main()
{ int dividendo, divisore, quoziente, resto;
  printf("inserisci il dividendo: ");
  scanf("%d",&dividendo);
  printf("inserisci il divisore: ");
  scanf("%d",&divisore);
  if (divisore == 0)
       printf("divisione impossibile");
   else
        quoziente = dividendo / divisore;
         resto = dividendo % divisore;
         printf("quoziente: %d\n",quoziente);
         printf("resto: %d\n",resto);

  }
```

Blocco all'interno di una selezione

```
#include <stdio.h>

main()
{ int dividendo, divisore, quoziente, resto;
  printf("inserisci il dividendo: ");
  scanf("%d",&dividendo);
  printf("inserisci il divisore: ");
  scanf("%d",&divisore);
  if (divisore != 0)
      {  quoziente = dividendo / divisore;
         resto = dividendo % divisore;
         printf("quoziente: %d\n",quoziente);
         printf("resto: %d\n",resto);
      }
   else printf("divisione impossibile");
}
```

Dichiarazioni in un blocco

```
#include <stdio.h>

main()
{ int dividendo, divisore;
  printf("inserisci il divi
  scanf("%d",&dividendo);
  printf("inserisci il divi
  scanf("%d",&divisore);
  if (divisore != 0)
      {  int quoziente, resto;
         quoziente = dividendo / divisore;
         resto = dividendo % divisore;
         printf("quoziente: %d\n",quoziente);
         printf("resto: %d\n",resto);
      }
   else printf("divisione impossibile");
}
```

Visto che le variabili quoziente e resto sono usate solo in uno dei due rami, posso definirle all'interno di questo blocco

Campo d'azione (scope)

- il **campo d'azione (scope)** di una variabile è la parte di programma in cui la variabile è nota e può essere manipolata
  - in C, Pascal: determinabile *staticamente. In C lo scope è il blocco in cui la variabile è definita*
  - in LISP: determinabile *dinamicamente*

scope di **x**

```
#include …
main()
{int x;
 …
}
```

Es.: scope delle variabili

```
#include <stdio.h>

main()
{ int dividendo, divisore;
  printf("inserisci il dividendo: ");
  scanf("%d",&dividendo);
  printf("inserisci il divisore: ");
  scanf("%d",&divisore);
  if (divisore != 0)
      {int quoziente, resto;
       quoziente = dividendo / divisore;
       resto = dividendo % divisore;
       printf("quoziente:%d",quoziente);
       printf(" resto:%d\n",resto);
      }
   else printf("divisione impossibile");
}
```

scope di quoziente e resto

scope di dividendo e divisore

Scope

- il concetto di scope diventerà importante quando studieremo le funzioni.
- In casi particolari, può essere utile definire le variabili nei blocchi interni
  - es, per risparmiare memoria (quando ho delle variabili che occupano molta memoria, che vedremo più avanti)

```
main()
{…
  if (condizione)
     { TipoCheOccupaMoltaMemoria1 var1;
       …
     }
  else
    { TipoCheOccupaMoltaMemoria2 var2;
       …
     }
}
```

  - Se avessi definito entrambe le variabili nel main, avrei avuto bisogno di una quantità di memoria pari alla somma

Scope: correzione errori

```
#include <stdio.h>

main()
{ int dividendo, divisore;
  printf("inserisci il dividendo:
  scanf("%d",&dividendo);
  printf("inserisci il divisore: "
  scanf("%d",&quoziente);
  if (divisore != 0)
       {  int quoziente, resto;
          quoziente = dividendo /
          resto = dividendo % divi
          printf("quoziente: %d\n"
          printf("resto: %d\n",res
       }
   else printf("divisione impossibile");
}
```

Mi sono sbagliato: ho usato la variabile `quoziente` invece di `divisore`: il compilatore mi dà errore.
Se avessi definito quoziente nel blocco main, il compilatore non mi direbbe dov'è l'errore e dovrei cercarlo a mano.

ESEMPIO di ISTRUZIONE IF

```
/* determina il maggiore tra due numeri */

#include <stdio.h>
main()
{
  int primo,secondo;

  scanf("%d%d",&primo,&secondo);
  if (primo >secondo)
       printf("%d",primo);
  else printf("%d",secondo);
}
```

ISTRUZIONI IF ANNIDATE

- Come caso particolare, <istruzione1> o <istruzione2> potrebbero essere un altro if
- Occorre attenzione *ad associare le parti else (che sono opzionali) all' if corretto*

Regola semantica:
l'**else** è sempre associato all'**if** più interno

```
if (n > 0)
  if (a>b) n = a;
  else n = b; /* riferito a if(a>b) */
```

Se vogliamo cambiare questa semantica, dobbiamo inserire un blocco

```
if (n > 0)
  { if (a>b) n = a; }
else n = b; /* riferito a if(n>0) */
```

ESEMPIO

Dati tre valori $a \le b \le c$ 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).

Vincolo: deve essere $c \le (a+b)$

• 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

Specifica:

se c ≤ (a+b)
  triangolo = vero
  se a==b==c { equil=isoscele=vero
            scaleno=falso }
  altrimenti
    se a==b o b==c o a==c { isoscele=vero;
                           equil=scaleno=falso }
    altrimenti
      { scaleno=vero;
        equil=isoscele=falso }
altrimenti
      triangolo = falso

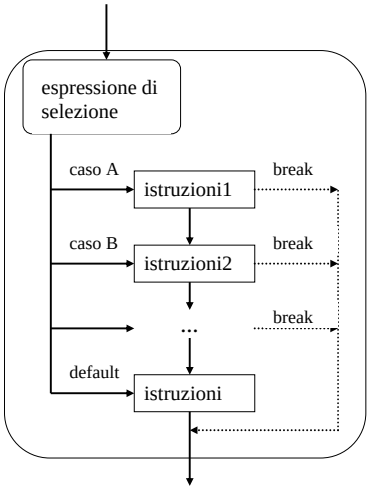
ESEMPIO

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

E` importante indentare correttamente per capire subito a quale if si riferisce un else

ISTRUZIONE DI SCELTA MULTIPLA

- Consente di scegliere fra *molte istruzioni (alternative o meno)* in base al valore di una *espressione di selezione*.
- L'espressione di selezione deve *denotare un valore numerabile* (intero o carattere).



ISTRUZIONE DI SCELTA MULTIPLA

```
<scelta-multipla> ::=
 switch (selettore)  {
 case <etichetta1> : <istruzioni> [break;]
 case <etichetta2> : <istruzioni> [break;]
          …
 [ default : < istruzioni>]
 }
```

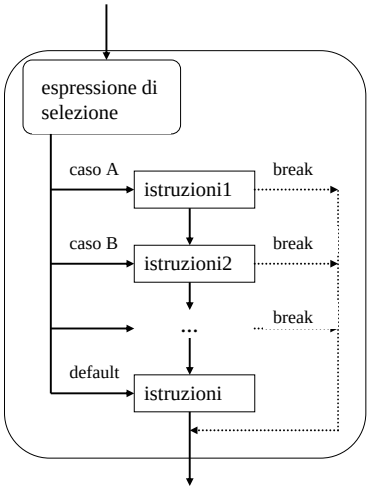
Sequenze, non occorre il blocco

Se nessuna etichetta corrisponde, si prosegue con il ramo **default** se esiste, altrimenti non si fa niente

- Il valore dell'espressione *selettore* viene confrontato con le etichette (costanti dello stesso tipo del selettore) dei vari casi: *l'esecuzione prosegue dal ramo corrispondente* (se esiste).

NOTA

I vari rami *non sono mutuamente esclusivi:* imboccato un ramo, si *eseguono anche tutti i rami successivi* a meno che non ci sia il comando **break** a *forzare esplicitamente* l'uscita.



ISTRUZIONE DI SCELTA MULTIPLA

```
switch (mese)
{
case 1 : giorni = 31; break;
case 2: if (bisestile) giorni = 29;
        else giorni = 28;
        break;
case 3:  giorni = 31;  break;
case 4:  giorni = 30;  break;
...
case 12:  giorni = 31;
}
```

ISTRUZIONE DI SCELTA MULTIPLA

- Alternativa

```
switch (mese)
{
case 2:
  if (bisestile) giorni = 29;
  else giorni = 28;
  break;
case 4:  giorni = 30;  break;
case 5:  giorni = 30;  break;
case 9:  giorni = 30;  break;
case 11:  giorni = 30;  break;
default:  giorni = 31;
}
```

Chi vuole esser milionario?

La mia banca mi dà un interesse del 5% annuo
Ora ho 1000€
Quanti anni mi servono per diventare milionario?

- Il totale inizialmente è 1000.
- Il numero di anni è 0
- Finché il totale è minore di 1000000, esegui
  - totale = totale * 1.05
  - anni = anni +1
- Stampa anni

CHI VUOL ESSER MILIONARIO?

- La mia banca mi dà un interesse annuo del 5%
- All'inizio ho 1000€
- Quanti anni devo aspettare per diventare milionario?

ALGORITMO

- Il totale inizialmente è 1000.
- Il numero di anni è 0
- Finché il totale è minore di 1000000, esegui
  - totale = totale*1.05
  - anni = anni+1
- Stampa anni

CHI VUOL ESSER MILIONARIO?

- La mia banca mi dà un interesse annuo del 5%
- All'inizio ho 1000€
- Quanti soldi avrò dopo 5 anni?

ALGORITMO

- Il totale inizialmente è 1000.
- Il numero di anni è 0
- Finché il numero di anni è minore di 5, esegui
  - totale = totale*1.05
  - anni = anni+1
- Stampa totale

DI CHE COSA HO BISOGNO?

- Devo eseguire più volte lo stesso codice.
- Eseguire un gruppo di istruzioni finché una condizione è vera
- Eseguire un gruppo di istruzioni per un numero prefissato di volte

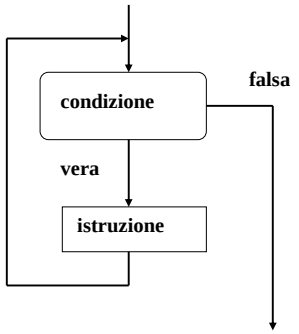
ISTRUZIONI DI ITERAZIONE

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

- Le istruzioni di iterazione:
  - hanno *un solo punto di ingresso* e *un solo punto di uscita* nel flusso del programma
  - perciò possono essere interpretate *come una singola azione* in una computazione sequenziale.

ISTRUZIONE `while`

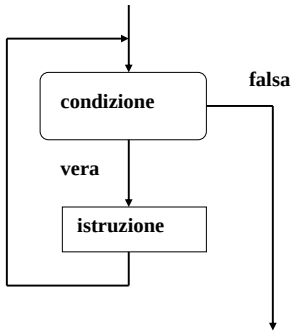
```
<while> ::=
    while(<condizione>) <istruzione>
```



- L'istruzione viene ripetuta *per tutto il tempo in cui la condizione rimane vera.*
- Se la condizione è falsa, l'iterazione non viene eseguita *neppure una volta.*
- In generale, *non è noto* ***quante volte*** l'istruzione sarà ripetuta.

ISTRUZIONE `while`

```
<while> ::=
    while(<condizione>) <istruzione>
```



Prima o poi, *direttamente o indirettamente*, l'istruzione deve *modificare la condizione*: altrimenti, l'iterazione durerà *per sempre! CICLO INFINITO*

Perciò, quasi sempre *istruzione è un blocco*, al cui interno si *modifica qualche variabile che compare nella condizione.*

CHI VUOL ESSER MILIONARIO?

- La banca mi dà un interesse annuo del 5%
- All'inizio ho 1000€
- Quanti anni devo aspettare per diventare milionario?

CHI VUOL ESSER MILIONARIO?

- La banca mi dà un interesse annuo del 5%
- All'inizio ho 1000€
- Quanti soldi avrò dopo 5 anni?

MEDIA DI N VOTI

- Si legga da tastiera un numero N
- Si leggano da tastiera N voti e se ne calcoli la media
- Si visualizzi la media dei voti

ESEMPIO ISTRUZIONE DI CICLO

```
#include <stdio.h>
main() /* Media di n voti*/
{ int    sum,voto,N,i;
  float  media;

  printf("Quanti sono i voti?");
  scanf("%d",&N);
  sum = 0;
  i = 1;
  while (i <= N)
   { printf("Dammi il voto n.%d:",i);
     scanf("%d",&voto);
     sum = sum+voto;
     i = i+1;
   }
  media=sum/N;
  printf("Risultato: %f",media);
}
```

MOLTIPLICAZIONE COME SEQUENZA DI SOMME

```
/* moltiplicazione come sequenza di somme */
#include <stdio.h>
main()
{
  int  X,Y,Z;

  printf("Dammi i fattori:");
  scanf("%d%d",&X,&Y);
  Z=0;
  while (X!=0)
        {  /* corpo ciclo while */
                   Z=Z+Y;
                   X=X-1;
        }
  printf("%d",Z);
}
```

ESEMPIO ISTRUZIONE DI CICLO

```
/* Calcolo del fattoriale di un numero N */

#include <stdio.h>
main()
{  int F, N, I;
   F=1;  /* inizializzazione del fattoriale*/
   I=1;  /* inizializzazione del contatore*/
   printf("Dammi N:");
   scanf("%d",&N);

   while (I <= N)
   {F = I*F;
    I = I+1;
   }
  printf("Il fattoriale e` %d", F);
}
```

CICLI A CONDIZIONE FINALE

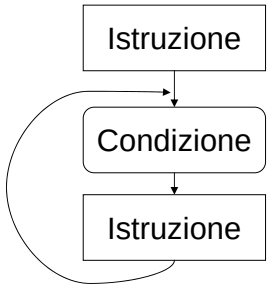
- Leggere un insieme di numeri e calcolarne la somma: non so a priori quanti sono i numeri
- Voglio chiedere all'utente i numeri e terminare la sequenza quando inserisce un codice speciale (ad esempio, 0)
- Algoritmo:

```
sum = 0;
stampa "inserisci un numero";
leggi num;
while (num != 0)
      { sum = sum + num;
        stampa "inserisci un numero";
        leggi num;
      }
stampa sum;
```

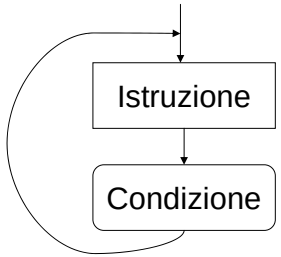
devo inserire in due punti le stesse istruzioni

CICLI A CONDIZIONE FINALE

- Se ho bisogno di eseguire l'istruzione prima di verificare la condizione, con il while devo riportare all'inizio l'istruzione

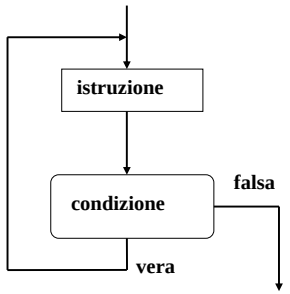


- In questo caso sarebbe più comodo un ciclo in cui prima eseguo l'istruzione e poi verifico la condizione



ISTRUZIONE `do..while`

```
<do-while> ::=
  do <istruzione>  while(<condizione>);
```



È una variante della precedente: la condizione viene verificata *dopo* aver eseguito l'istruzione.

Se la condizione è falsa, l'iterazione **viene comunque eseguita *almeno una volta***.

ESEMPIO ISTRUZIONE DI CICLO

```
/* Calcolo della somma di N numeri, N non
 noto a priori */

#include <stdio.h>
main()
{  int sum, num;
   sum = 0;
   num = 0;
   do
   { sum = sum + num;
     printf("Dammi un numero (0 termina) ");
     scanf("%d",&num);
   } while (num != 0);
   printf("La somma e` %d", sum);
}
```

ESEMPIO ISTRUZIONE DI CICLO

```
/* Calcolo del fattoriale di un numero N */

#include <stdio.h>
main()
{  int F, N, I;
   F=1;  /* inizializzazione del fattoriale*/
   I=1;  /* inizializzazione del contatore*/
   printf("Dammi N:");
   scanf("%d",&N);
   do
   {  F = I*F;
      I = I+1;
   }
   while (I <= N);
   printf("Il fattoriale e' %d", F);
}
```

ESEMPIO

- Nell'istruzione **while**, la condizione di ripetizione viene verificata **all'inizio di ogni ciclo**

```
...
somma=0; j=1;
while (j <= n)
  { somma = somma + j;  j++; }
```

- Nell'istruzione **do** la condizione di ripetizione viene verificata **alla fine di ogni ciclo**

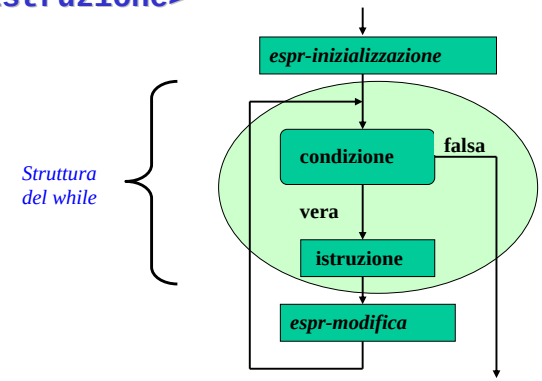
```
/* In questo caso: n > 0  */
somma = 0; j = 1;
do
  { somma = somma + j;  j++; }
while (j <= n);
```

ISTRUZIONE `for`

- È una evoluzione dell'istruzione **`while`** che mira a eliminare alcune frequenti sorgenti di errore:
  - mancanza delle *inizializzazioni delle variabili*
  - mancanza della *fase di modifica del ciclo* (rischio di ciclo senza fine)
- In genere si usa quando è noto il numero di volte in cui dovrà essere eseguito il ciclo.

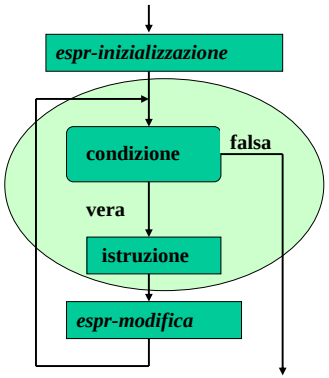
ISTRUZIONE for

```
<for> ::=
 for( <espr-iniz>;<cond>;<espr-modifica>)
 <istruzione>
```



ISTRUZIONE for

```
<for> ::=
 for(<espr-iniz>;<cond>;<espr-modifica>)
 <istruzione>
```



Espressione di inizializzazione: **`<espr-iniz>`**
valutata *una e una sola volta* *prima* di iniziare l'iterazione.

Condizione: **`<cond>`**
valutata *a ogni interazione*, per decidere se proseguire (come in un while). Se manca si assume *vera*!

Espressione di modifica: **`<espr-modifica>`**
valutata *a ogni interazione*, *dopo* aver eseguito l'istruzione.

ESEMPIO ISTRUZIONE DI CICLO

```
#include <stdio.h>
main() /* Media di n voti*/
{ int    sum,voto,N,i;
  float  media;

  printf("Quanti sono i voti ?");
  scanf("%d",&N);
  sum = 0;
  for(i = 1; i <= N;i++)
   { printf("Dammi il voto n.%d:",i);
     scanf("%d",&voto);
     sum=sum+voto;
   }
  media = sum/N;
  printf("Risultato: %f",media);
}
```

Nota: non serve l'inizializzazione del contatore `i` e l'incremento di `i` nel ciclo

RIPRENDIAMO IL CASO DEL WHILE

```
#include <stdio.h>
main() /* Media di n voti*/
{ int    sum,voto,N,i;
  float  media;

  printf("Quanti sono i voti ?");
  scanf("%d",&N);
  sum = 0;
  i = 1;
  while (i <= N)
   { printf("Dammi il voto n.%d:",i);
     scanf("%d",&voto);
     sum=sum+voto;
     i=i+1;
   }
  media = sum/N;
  printf("Risultato: %f",media);
}
```

ESEMPIO ISTRUZIONE DI CICLO

```
/* Calcolo del fattoriale di un numero N */
#include <stdio.h>
#include <math.h>
main()
{
  int     N, F, I;

  printf(“Dammi N:”);
  scanf("%d",&N);
  F=1;        /*inizializzazione del fattoriale*/
  for (I=2;I <= N; I++)
    F=F*I;

  printf("Fattoriale: %d",F);
}
```

RIPRENDIAMO IL CASO DEL WHILE

```
/* Calcolo del fattoriale di un numero N */

#include <stdio.h>
main()
{  int F, N, I;
   F=1;  /* inizializzazione del fattoriale*/
   I=1;  /* inizializzazione del contatore*/
   printf(“Dammi N:”);
   scanf("%d",&N);

   while (I <= N)
   {F = I*F;
    I = I+1;
   }
  printf(“Il fattoriale e’ %d”, F);
}
```

ESEMPIO

- Dati due valori positivi *X* e *Y*, calcolarne la divisione intera X/Y come sequenza di sottrazioni, ottenendo quoziente e resto.

Invariante di ciclo:

$$X = Q * Y + R, \quad \text{con } R \geq 0$$

- inizialmente, Q=0, R=X *(R>Y)*
- a ogni passo, Q’=Q+1, R’=R-Y *(R>Y)*
- alla fine, $X = Q^{(n)} * Y + R^{(n)}$ *(0≤R<Y)*
  che è la definizione di divisione intera.

ESEMPIO

Specifica:

sia *Q* il quoziente, inizialmente pari a 0
sia *R* il resto, inizialmente pari a *X*
while (*R* ≥ *Y*)
  incrementare il quoziente *Q*
  decrementare *R* di una quantità *Y*

Codifica

```
main(){
  int x = 20, y = 3, q, r;
  for (q=0, r=x;  r>=y;  q++, r=r-y);
}
```

Idem per l’espressione di modifica

Notare l’uso di una espressione concatenata per concatenare due assegnamenti e inizializzare così due variabili.

Esercizi

- Verificare se un numero è primo
- Stampare la tavola pitagorica
- Calcolare la fattorizzazione di un numero
- Calcolare tutte le terne pitagoriche in cui il lato è inferiore a 500
- Calcolare la potenza x^n con moltiplicazioni successive
- Calcolare la funzione seno di *x* con la seguente formula:

$$sen(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \ldots$$

Ancora sulla programmazione strutturata

- L'idea della programmazione strutturata è basata sul fatto che ogni costrutto ha solo un punto di ingresso ed uno di uscita
- Questo significa che
  - i costrutti possono essere considerati come macro-istruzioni
  - è facile capire sotto quali condizioni si esce dall'istruzione
  - posso modificare l'istruzione essendo abbastanza sicuro che non ci saranno grosse ripercussioni in parti non previste del programma
- In C esistono anche istruzioni che non seguono questa filosofia: noi non le usiamo!!!