Strutture iterative

Ver. 3

© 2021 - Claudio Fornaro - Corso di programmazione in C (C89)

Strutture iterative

- Problema:
  Visualizzare i numeri interi da 0 a 1000

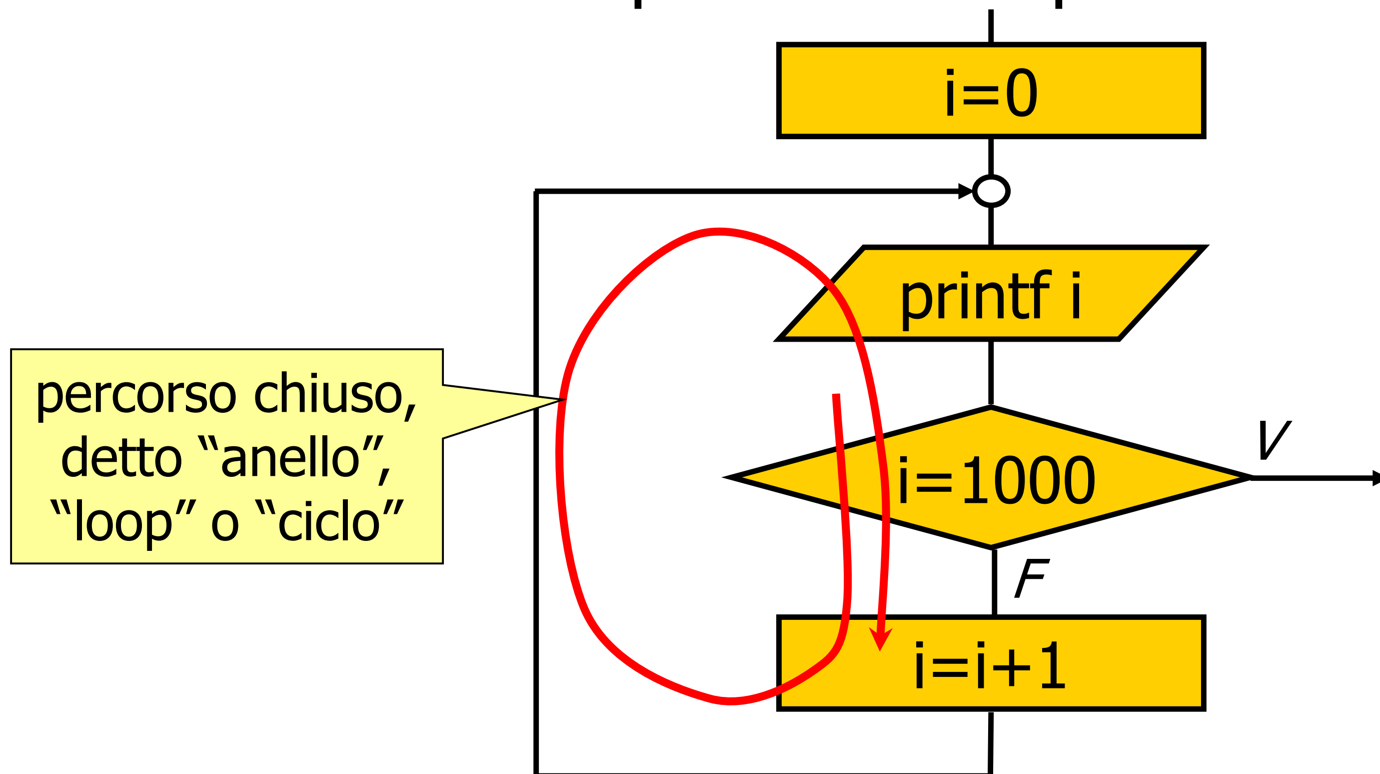
- Soluzione:
  ```
  printf("0\n");
  printf("1\n");
  printf("2\n");
  printf("3\n");
  printf("4\n");
  …
  ```
  Sono 1001 numeri e non è davvero una buona idea, ma con le conoscenze attuali non c’è alternativa

Strutture iterative

- La soluzione sarebbe poter scrivere:
  "*Ripeti l'istruzione:*
  ```
  printf("%d\n", i);
  ```
  con `i` *che va da 0 a 1000"*
- Ossia seguire i seguenti passi:
  - per partire da 0 serve un'istruzione `i=0` all'inizio
  - poi deve essere eseguita la `printf("%d\n", i);`
  - quindi si valuta `i` e se non è 1000:
    - incrementa `i` di 1
    - *torna indietro* alla `printf` per continuare
  - Altrimenti si è finito

Strutture iterative

- Il flow chart corrispondente è questo:



- È corretto, ma dal punto di vista formale si vedrà che non è *strutturato* e sarà quindi necessario modificarlo (di poco)

Strutture iterative

- La traduzione diretta in C del flow chart precedente richiede l’utilizzo di un’istruzione specifica per “tornare indietro”, il C ha questa l’istruzione che si chiama `goto`, ma per motivi che saranno descritti più avanti è da evitare, *approssimativamente* il codice C sarebbe:

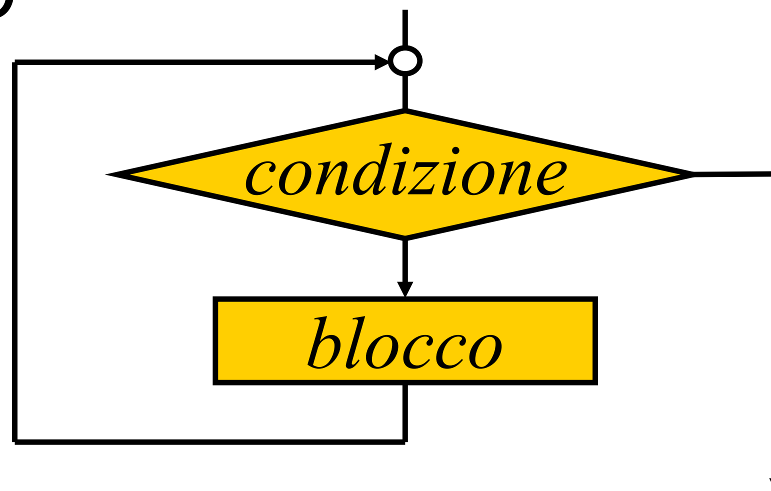
```
i=0;
printf("%d\n", i);
if (i!=1000)
{
    i++;
    goto…;        va alla printf
}
```

Strutture iterative

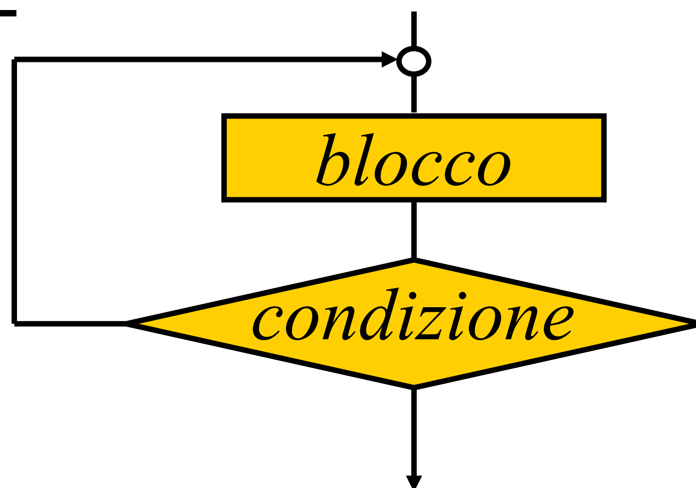
- Per evitare il `goto` e garantire la possibilità di scrivere sempre *codice strutturato* (concetto descritto in seguito) sono necessarie strutture sintattiche specifiche per far eseguire più volte un *blocco* di righe di codice (detto **corpo del ciclo**)
- Queste strutture sintattiche sostanzialmente sono due e sono composte di *due soli blocchi:* uno di controllo e uno di esecuzione (il flow chart precedente invece aveva tre blocchi nel ciclo); a seconda di come sono composti i due blocchi si hanno **cicli while** e **cicli do-while**

Strutture iterative

- Ciclo WHILE-DO

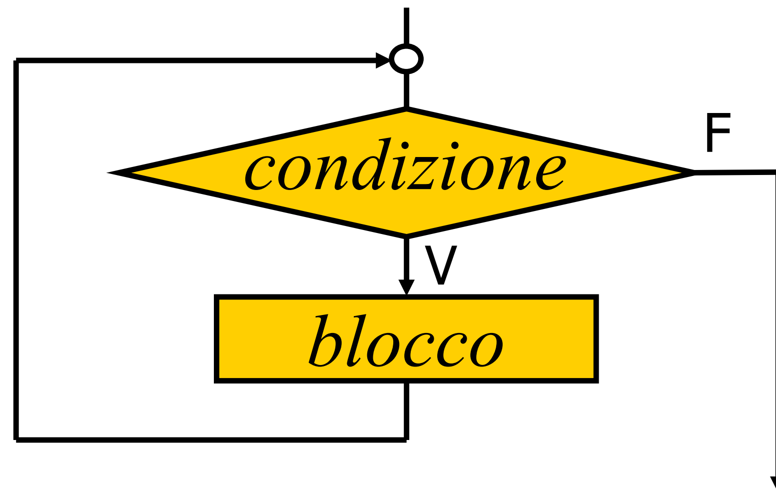


- Ciclo DO-WHILE



Ciclo WHILE

- In un ciclo WHILE la *condizione* viene valutata **prima** di eseguire il *blocco*
- Il *blocco* viene eseguito *fintantoché* la *condizione* è vera



- Se la *condizione* è inizialmente falsa, il *blocco* non viene eseguito neppure una volta

Ciclo WHILE

- Sintassi:

  `while (`*condizione***`)`** ← *senza il ';'*

  blocco
- Viene valutata la *condizione*:
  - *se è vera*
    - esegue il *blocco*
    - torna automaticamente su a valutare la *condizione*
  - *se è falsa*
    - passa ad eseguire le istruzioni successive a *blocco*
- La *condizione* è un'espressione qualsiasi (come quella del costrutto `if`) e non può mancare
- Per migliorare la leggibilità non si mettano spazi tra `while` e la parentesi

Ciclo WHILE

- Esempio
  Il seguente codice risolve il problema iniziale di visualizzare i numeri interi da 0 a 1000

```
i=0;
while (i<=1000)
{
  printf("%d ", i);
  i++;
}
```

- Dopo che è terminato il ciclo, `i`=1001: infatti dopo aver visualizzato 1000, `i++` incrementa `i` a 1001 che non soddisfa più la *condizione*

Ciclo WHILE

- Esempio 2
  Sommare i valori introdotti dalla tastiera finché non viene immesso il valore 0

```
somma = 0;
scanf("%d", &v);      ← primo valore
while (v != 0)
{ somma += v;
  scanf("%d", &v);    ← successivi valori
}
printf("Somma: %d", somma);
```

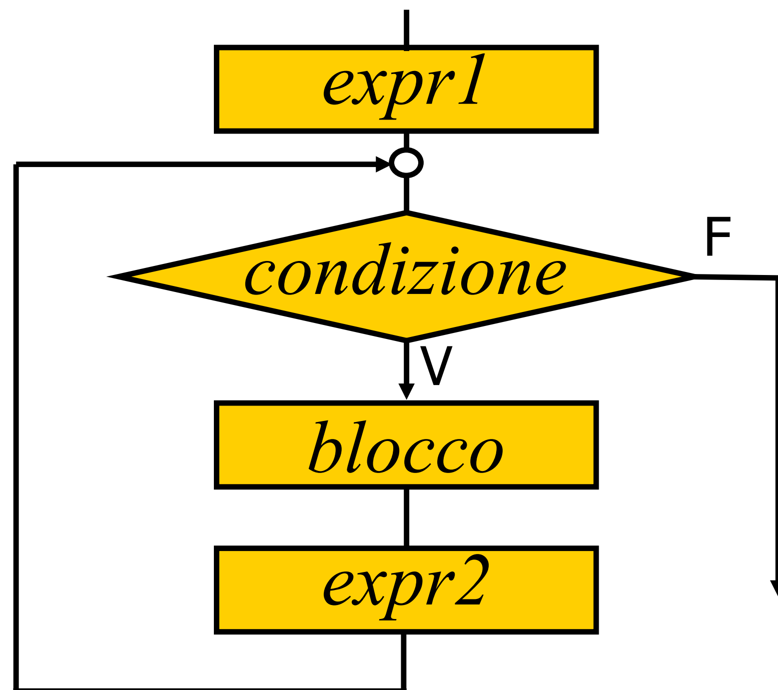
  - La `scanf` nel corpo del ciclo preleva i valori successivi al primo
  - Una variabile che accumula valori come `somma` è detta *accumulatore*

Ciclo FOR

- Poiché il ciclo WHILE viene spesso usato con una variabile che viene verificata e aggiornata (in genere incrementata) a ogni iterazione, il C fornisce una riscrittura compatta del WHILE che mette in evidenza in un’unica riga l’inizializzazione, il controllo e l’aggiornamento della variabile stessa
- Questa riscrittura compatta è il ciclo FOR

Ciclo FOR

- Il flow-chart di un ciclo FOR è il seguente:



in cui:

- *expr1* è l'inizializzazione della variabile (es. `i=0`)
- *expr2* è l'aggiornamento della variabile (es. `i++`)

Ciclo FOR

- In un ciclo `for` il *blocco* viene eseguito *fintantoché* la *condizione* è vera (come `while`):

```
for (expr1; condizione; expr2)   ← senza il ';'
    blocco
```

- Viene calcolata *expr1* soltanto la prima volta
- Viene valutata la *condizione*:
  - *se è vera:*
    - esegue il *blocco*
    - esegue *expr2*
    - torna su a valutare nuovamente la *condizione*
  - *altrimenti (se è falsa):*
    - passa ad eseguire le istruzioni successive a *blocco*
- Per migliorare la leggibilità non si mettano spazi tra `for` e la parentesi

Ciclo FOR

- La *condizione* è un'espressione qualsiasi (come quella del costrutto `if`) e può mancare, in questo caso viene considerata 1 (ossia vera)
- *expr1* e/o *expr2* possono mancare (ma i separatori ';' devono esserci sempre entrambi)
- *expr1* ed *expr2* sono generiche espressioni, solitamente *expr1* è una semplice inizializzazione, ma è frequente che *expr2* sia più complessa di un semplice incremento:

```
for (i=a*b-5; i<c*d; i=a*i+2) …
```

Ciclo FOR

- Esempio
  Anche il seguente codice risolve il problema iniziale di visualizzare i numeri interi da 0 a 1000, come si vede è molto più compatto:

```
for (i=0; i<=1000; i++)
    printf("%d ", i);
```

- Una variabile che tiene conto del numero di iterazioni, come la `i` in questo esempio, viene detta ***variabile di conteggio*** o ***indice***
- Si noti che, nell’esempio, dopo che il ciclo è stato eseguito completamente `i` vale 1001 (come nell’esempio del ciclo `while`)

Ciclo FOR

- La ripetizione "*per N volte*" si può ottenere in diversi modi; esempi con `i` crescenti:
  - `for (i=0; i<N; i++)`
    qui `i` va da 0 a N–1, N *escluso*
    quando esce dal ciclo, `i` vale N
  - `for (i=1; i<=N; i++)`
    qui `i` va da 1 a N *incluso*
    quando esce dal ciclo, `i` vale N +1
  - `for (i=0; i<=N; i++)`
    Attenzione: qui il ciclo viene eseguito N+1 volte, `i` va da 0 a N *incluso*; quando esce, `i` vale N+1
  - `for (i=23; i<45; i++)`
    qui `i` va da 23 a 44, 45 *escluso*;
    quando esce dal ciclo, `i` vale 45

Ciclo FOR

- La ripetizione "*per N volte*" si può avere anche con `i` decrescenti, ad esempio:
  - `for (i=N; i>0; i--)`
    qui `i` va da N a 1, 0 *escluso*
    quando esce dal ciclo, `i` vale 0
  - `for (i=N-1; i>=0; i--)`
    qui `i` va da N–1 a 0 *incluso*
    quando esce dal ciclo, `i` vale –1
  - `for (i=N; i>=0; i--)`
    Attenzione: qui il ciclo viene eseguito N+1 volte, `i` va da N a 0 *incluso*; quando esce, `i` vale –1
  - `for (i=45; i>23; i--)`
    qui `i` va da 45 a 24, 23 *escluso*;
    quando esce dal ciclo, `i` vale 23

Ciclo FOR

- I due cicli FOR e WHILE seguenti sono equivalenti, si noti la presenza e la disposizione degli stessi elementi che li compongono

```
for (expr1; condizione; expr2)
    blocco


expr1;          ← fuori dal corpo del ciclo!
while (condizione)
{
    blocco
    expr2;
}
```

Ciclo FOR

- Esempio
  Questo ciclo WHILE:

```
i=0;
while (i<=1000)
{
  printf("%d", i);
  i++;
}
```

  e questo ciclo FOR:

```
for (i=0; i<=1000; i++)
  printf("%d", i);
```

  sono equivalenti, ma il secondo è più compatto

La variabile di conteggio

- Per questioni di efficienza è preferibile che la variabile di conteggio sia di tipo intero
- È spesso conveniente che il nome della variabile di conteggio sia corto (tipicamente `i`, `j`, `k`, ...) per migliorare la *leggibilità* del codice
- Esempio

```
for (i=0; scanf("%d",&v[i])!=EOF; i++)
  tot += v[i]*v[i-1]*v[i+1];
numeroValoriLetti = i;
```

Qui `i` viene usata più volte nel ciclo, usando `i` il codice è più leggibile rispetto a quello che si avrebbe usando invece `numeroValoriLetti`, questa comunque viene assegnata a fine ciclo

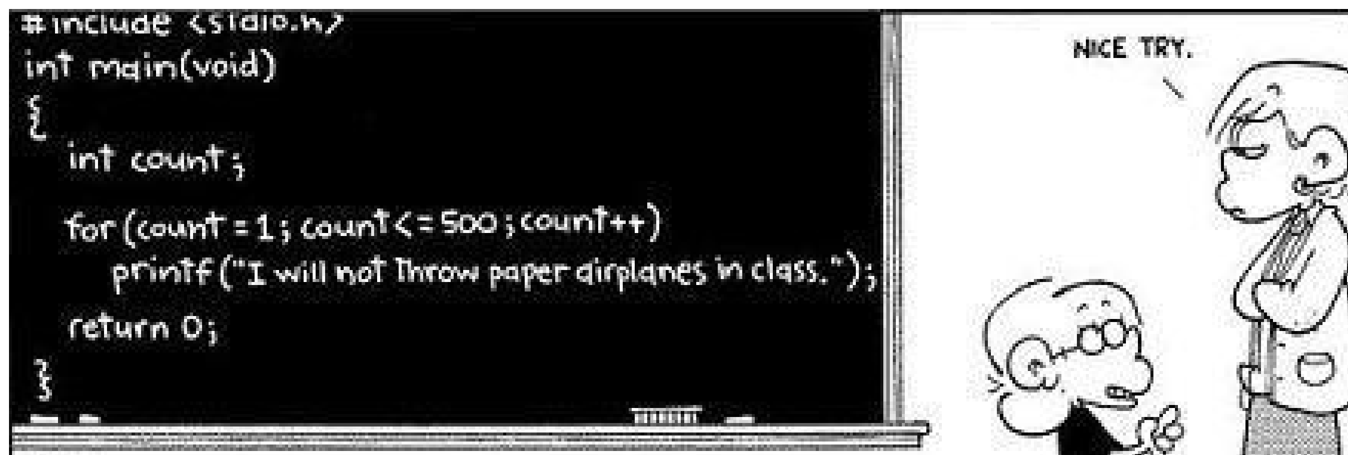
Scelta tra ciclo FOR e WHILE

- Quando il numero di iterazioni non è noto a priori, è *preferibile* usare un ciclo WHILE (anche in altri linguaggi il WHILE è un ciclo controllato da una condizione)
- Quando il numero di iterazioni è noto a priori, per chiarezza e stilisticamente è *preferibile* utilizzare un ciclo FOR in quanto raggruppa in un punto solo la gestione dell'indice
- In altri linguaggi il ciclo FOR è un **vero** *ciclo a conteggio*, ossia la sua terminazione non si ha valutando una condizione, ma raggiungendo il valore finale specificato

Scelta tra ciclo FOR e WHILE

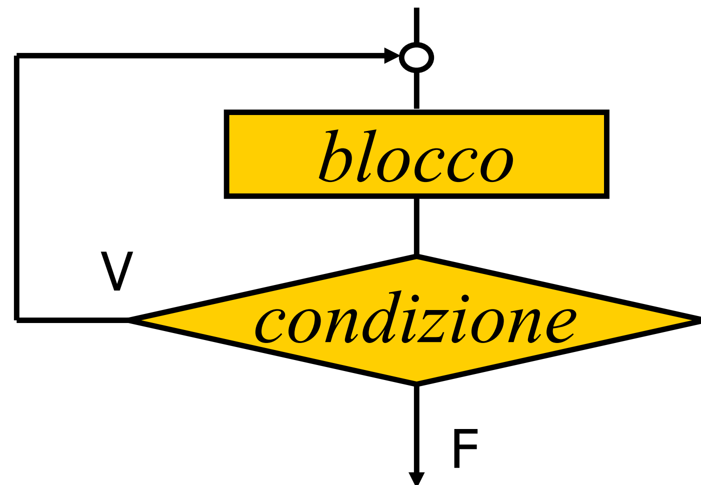
- Conviene usare un ciclo FOR anche quando si devono *contare le iterazioni* anche se tale numero non è noto a priori, in questi casi la terminazione del ciclo dipende da una condizione che non riguarda l'indice

```
for (i=0; (c=getchar()) == ' '; i++)
  ;
```



Ciclo DO-WHILE

- In un ciclo DO-WHILE la *condizione* viene valutata **dopo** aver eseguito il *blocco*
- Il *blocco* viene eseguito *fintantoché* la *condizione* è vera



- Anche se la *condizione* è inizialmente falsa, il *blocco* viene eseguito almeno una volta

Ciclo DO-WHILE

- Sintassi

```
do
{
    blocco
}while (condizione);
```
← *richiede il ';'*

- Le graffe sono opzionali, ma *consigliabili*. La graffa di chiusura messa subito prima della keyword `while` permette di distinguere facilmente il ciclo WHILE dal ciclo DO-WHILE, lo spazio tra `while` e parentesi è per leggibilità
- Altri linguaggi hanno il *ciclo Repeat-Until* che ha la sola differenza di terminare quando la condizione *diventa* vera

Ciclo DO-WHILE

- Esempio
  Anche il seguente codice risolve il problema iniziale di visualizzare i numeri interi da 0 a 1000

```
i=0;
do
{
  printf("%d ", i);
  i++;
}while (i<=1000);
```

- Usciti dal `while` il valore di `i` è 1001

Ciclo DO-WHILE

- Esempio 2
  Somma i valori dati finché non viene introdotto il valore 0

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

Notare che il valore `v` viene comunque addizionato a `somma` (ma in questo esempio non causa problemi perché somma 0)

Scelta tra ciclo WHILE e DO

- La scelta tra ciclo WHILE e ciclo DO-WHILE è spesso ovvia, in altri casi è solo questione di preferenze personali
- Si può sempre passare da un tipo di ciclo all’altro modificando (di poco) il programma

Corpo di un ciclo

- Quando in un ciclo tutta l'elaborazione è già contenuta nella *condizione* e/o nelle *expr1* ed *expr2* e non serve che il corpo contenga altre istruzioni, poiché il corpo deve comunque esistere, si usa un'*istruzione nulla o istruzione vuota*, ossia il solo carattere ';'
- Per chiarezza è bene sia collocato, indentato, in una riga vuota, MAI in fondo a `for` o `while`

```
for (i=0; scanf("%d", &v)!=EOF; i++)
    ;
```

- In alternativa si possono usare una coppia di parentesi graffe `{}` o l'istruzione `continue`

Programmazione strutturata

- Il Linguaggio C è un linguaggio *imperativo* (si indicano i singoli passi di base da eseguire per ottenere il risultato voluto) *procedurale* (il codice sorgente è strutturato a blocchi)
- La programmazione strutturata è una metodologia di programmazione che essenzialmente limita l’organizzazione di questi blocchi al fine di migliorare la chiarezza e la manutenibilità del codice

Programmazione strutturata

- Un linguaggio strutturato deve avere almeno i seguenti 3 tipi di strutture di controllo:
  - **sequenza:** permette di definire un blocco composto da più istruzioni
  - **selezione:** permette di decidere se eseguire o no un blocco di codice
  - **ciclo:** permette di per ripetere uno stesso blocco di codice più volte
- Inoltre ogni struttura di controllo (includendo il blocco controllato) deve avere un unico punto di ingresso (*entry point*) e un unico punto di uscita così da poter essere considerata essa stessa un'unica *(macro-)istruzione*

Programmazione strutturata

- Un programma è strutturato se è la composizione di queste sole strutture
- Ogni struttura può contenere altre strutture, ad esempio una sequenza può contenere un ciclo il cui blocco è a sua volta una sequenza contenente costrutti di selezione di altre strutture
- Ogni programma non strutturato può essere modificato in modo che sia strutturato

Programmazione strutturata

- Il C è un linguaggio strutturato in quanto dispone di tutte le strutture necessarie per scrivere codice completamente strutturato:
  - **sequenza:** una semplice successione di istruzioni costituisce una sequenza, queste istruzioni possono anche essere raggruppate in una macro-istruzione (blocco) per mezzo di *parentesi graffe*
  - **selezione:** i costrutti `if` e `switch`
  - **ciclo:** i costrutti `for`, `while` e `do-while`
- Diversamente da altri linguaggi, il C permette di scrivere anche codice non strutturato, ma è buona norma **evitarlo**

Programmazione strutturata

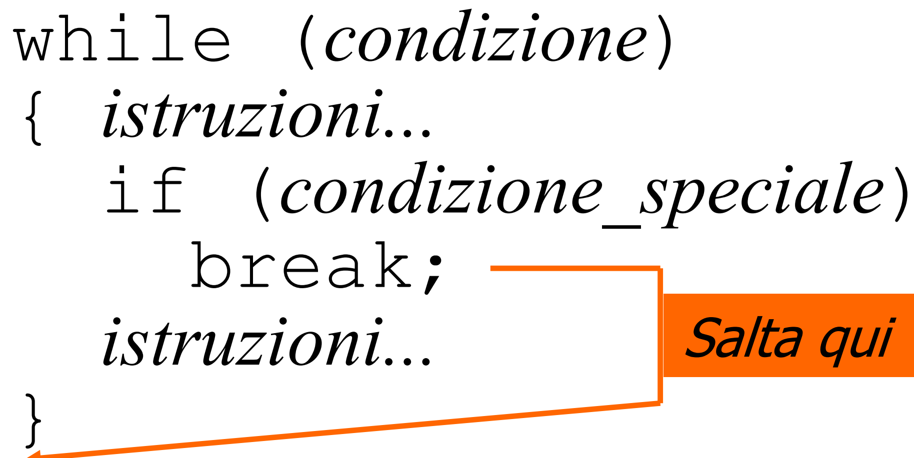
- In Linguaggio C si ha programmazione NON strutturata quando si usano le istruzioni:
  - `goto`
  - `break` e `continue`
  - più di una `return`/`exit` in una stessa funzione (`main` incluso)
- L'*abuso* di queste istruzioni, in particolare la `goto`, *può* portare a codice poco chiaro, ma è innegabile che talvolta, se usate con competenza, diano vantaggi anche importanti in termini di chiarezza del codice e velocità di esecuzione (vedere più avanti)

Programmazione strutturata

- Se ci si vuole attenere a una programmazione *puramente* strutturata, non si devono usare le istruzioni indicate
- Dal punto di vista didattico è la sola `goto` che è bene evitare

Break

- Fa uscire immediatamente da un ciclo
- Nel caso di ciclo `for`, non viene eseguita *expr2*, ossia l' aggiornamento della variabile di ciclo
- Dopo il `break`, l'esecuzione continua dalla prima riga *successiva* al blocco



- Il break viene di norma usato per gestire condizioni particolari (ma non deve essere il metodo normale di terminazione di un ciclo)

Break

- Esempio
  Calcola la media di al massimo 10 valori dati in input, ma uno 0 lo fa terminare in anticipo.

```
double v, somma = 0;
for (i=0; i<10; i++)
{
   scanf("%lf", &v);
   if (v == 0)
      break;
   somma += v;
}
printf("Media = %f\n", somma/i);
```

Break

- La formulazione strutturata equivalente è:

```
int esci = NO;  ← esempio di variabile flag
double v, somma = 0;
for (i=0; i<10 && esci==NO; i++)
{
   scanf("%lf", &v);
   if (v == 0)
      esci = SI;
   else
      somma += v;
}
i--;        ← i è stata comunque incrementata
printf("Somma = %f\n", somma/i);
```

Continue

- Fa passare immediatamente all'iterazione successiva
- Per effetto dell'istruzione `continue`:
  - vengono saltate tutte le istruzioni dopo la `continue` fino fine del corpo del ciclo
  - se si tratta di un ciclo `for`, viene comunque eseguita *expr2*
  - l'esecuzione riprende dalla valutazione della *condizione*

Continue

- Schema con ciclo `while`

```
while (condizione)
{
    istruzioni...

    if (condizione_speciale)
        continue;

    altre istruzioni, saltate se è eseguita continue
}
```

Continue

- Schema con ciclo `for`

```
for (espr1; condizione; espr2)
{
    istruzioni...

    if (condizione_speciale)
        continue;

    altre istruzioni, saltate se è eseguita continue
}
```

Continue

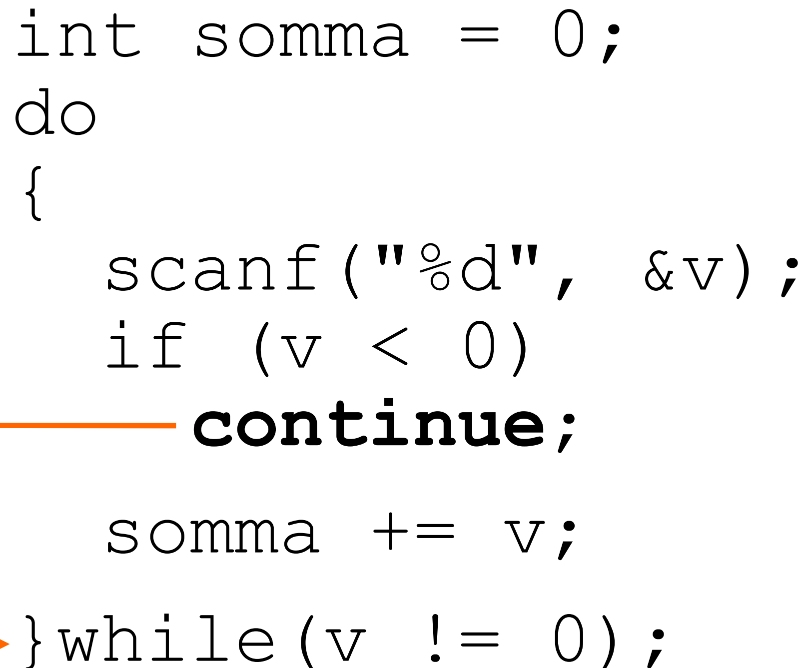
- Schema con ciclo `do-while`

```
do
{
   istruzioni...

   if (condizione_particolare)
      continue;

   altre istruzioni, saltate se è eseguita continue

}while (condizione);
```

Salta qui

Continue

- Esempio
  Somma i valori dati finché non viene introdotto 0, ignorando i valori negativi.



```
int somma = 0;
do
{
  scanf("%d", &v);
  if (v < 0)
    continue;

  somma += v;

}while(v != 0);
```

Continue

- Invertendo il confronto nella `if` si ha una formulazione più chiara e anche strutturata, qui conviene sicuramente farlo:

```
int somma = 0;
do
{
    scanf("%d", &v);
    if (v >= 0)
        somma += v;
}while(v != 0);
```

Continue

- Attenzione nella trasformazione di un ciclo FOR in WHILE (o viceversa) quando si hanno istruzioni `continue`: negli esempi seguenti la `i` viene incrementata diversamente

```
sum=0;
for (i=0;i<10;i++)
{
   scanf("%d",&v);
   if (i == 0)
      continue;
   sum += v;
}
```

≠

```
sum=0; i=0;
while (i<10)
{
   scanf("%d",&v);
   if (i == 0)
      continue;
   sum += v;
   i++; saltata dal continue
}
```

Lettura di valori

- Quando non è noto a priori il numero di valori che verranno introdotti dall'utente, si deve trovare un modo alternativo per indicare al programma che non ci sono più dati in input
- Ad esempio si potrebbe pensare che il seguente ciclo termini quando si dà Invio invece di introdurre un valore:

```
while (scanf("%d", &x) == 1)
```

  Non funziona in quanto la specifica `%d` elimina tutti i white spaces iniziali e premendo Invio si dà semplicemente un altro white space che la funzione continua a leggere ed eliminare

Lettura di valori

- Un modo per identificare la terminazione dell'input quando non si sa a priori il numero di dati che saranno immessi fa uso del concetto di *sentinella*
- La *sentinella* è un valore particolare (es. lo 0 negli esempi precedenti) che indica la fine della sequenza:

```
fine = 0;  ← questa è la sentinella
while (scanf("%d", &x) && x!=fine)
{
   ...
```

Valutata dopo l'acquisizione di x

- La sentinella è riconosciuta dopo l'acquisizione

Lettura di valori

- Altro modo per indicare la fine dei dati in input è introdurre su una riga vuota il *carattere di fine input*, detto EOF (End Of File)
- Quando le funzioni di input leggono un EOF riportano che non ci sono più dati da leggere e restituendo valori speciali:
  - le funzioni che restituiscono un valore intero (es. la `scanf` e la `getchar`) restituiscono il *valore* `EOF` (attenzione che il *valore* `EOF` NON È il codice ASCII del *carattere* EOF, ma un valore numerico definito in `<stdio.h>` e tipicamente pari a –1)
  - le funzioni che restituiscono un puntatore (es. la `gets`) restituiscono il valore `NULL`

Lettura di valori

- Si ricorda che per introdurre il carattere EOF:
  - *Windows* → premere Control-Z e poi Invio
  - *Linux/Unix/OSX* → premere Control-D
- Esempio

```
while (scanf("%d", &a) != EOF)
    somma += a;
printf("Somma = %d\n",somma);
```

Esecuzione (Windows):

12*Invio*
51*Invio*
34*Invio*
Control-Z Invio
Somma = 97

Esecuzione (Unix/Linux):

12*Invio*
51*Invio*
34*Invio*
Control-D
Somma = 97

Lettura di valori

- Esempio

```
while (scanf("%d", &a) != EOF)
    somma += a;
printf("Somma = %d\n",somma);
```

Esecuzione (sotto Windows):

12*Invio*
51*Invio*
34*Invio*
Control-Z *Invio*
Somma = 97

Lettura di valori

- Se si vuole terminare la lettura quando si dà un Invio su una riga vuota bisogna controllare se c’è un `\n` su una riga vuota
- L’esempio che segue permette di inserire uno o più valori separati da spazi su ciascuna riga e terminare con un Invio su una riga vuota

```
while ((c=getchar())!= '\n')
{
    ungetc(c, stdin);
    scanf("%d%*c", &a);
    …
}
```

Lettura di valori

- Per comprendere il codice precedente è necessario conoscere i file e le stringhe
- Nell'esempio la `getchar` legge un carattere e se è un `\n` esce dal ciclo, altrimenti usa la funzione `ungetc` (slide sui file) per rimettere il carattere in `stdin` (ossia la tastiera) come se non lo avesse letto (altrimenti la `scanf` leggerebbe solo dal secondo carattere)
- La `scanf` legge con `%d` un valore intero e poi il carattere subito a destra, questo è o lo spazio tra due numeri sulla stessa riga o il `\n` di fine riga, usa `%*c` per scartarlo

Cicli annidati

- Un ciclo può essere *interamente* collocato (*annidato*) nel corpo di un altro ciclo (insieme ad altre istruzioni se necessario)
- Il ciclo esterno *controlla* quello interno
- Nel caso di cicli FOR ogni ciclo deve avere una variabile di conteggio diversa, altrimenti il ciclo interno modificherebbe l’indice di controllo del ciclo esterno
- Ogni volta che il ciclo esterno esegue un’iterazione, il ciclo interno ricomincia da capo (dall’inizializzazione del suo indice)

Cicli annidati

- Esempio

ciclo esterno

ciclo interno

```
for (i=1; i<=7; i+=3)
{
   for (j=2; j<5; j++)
     printf("%d,%d ", i, j);
   printf("\n");
}
printf("Val. finali:%d,%d ", i, j);
```

Produce il seguente output:

```
1,2 1,3 1,4
4,2 4,3 4,4
7,2 7,3 7,4
Val. finali:10,5
```

← *notare i valori finali*

Uscita da cicli annidati

- `break` fa uscire solo dal FOR corrente; nel caso di cicli annidati, per uscire da tutti i cicli annidati **si può** (opinione personale) considerare di usare un `goto`

```
for (i=0; i<10; i++)
  for (j=0; j<10; j++)
  { scanf("%d", &v);
    if (v == 0)
      goto fuori;
    somma += v;
  }
fuori:
printf("Somma = %d\n",somma);
```

Si noti che `i` e `j` NON sono stati incrementati

Uscita da cicli annidati

- Per evitare di avere codice non strutturato e a scapito di un po' di efficienza si può scrivere:

```
esci = NO;           ← variabile flag
for (i=0; i<10 && esci==NO; i++)
  for (j=0; j<10 && esci==NO; j++)
  { scanf("%d", &v);
    if (v == 0)
      esci = SI;
    else
      somma += v;
  }
printf("Somma = %d\n",somma);
```

Si noti che `i` e `j` sono stati incrementati

Variabili flag

- Una variabile *flag* (“bandierina di segnalazione”) serve a indicare che il programma è in uno stato specifico (nel caso precedente indica se uscire o no dal ciclo)
- È tipicamente una variabile booleana
- Normalmente viene impostata a un valore iniziale e cambiata nel valore opposto quando capita un *evento particolare* (nel caso precedente `v==0`), in genere non deve essere nuovamente riportata al valore iniziale (es. se l’evento non si verifica), altrimenti non si può sapere se l’evento è avvenuto o no

Etichette

- Una *label* (*etichetta*) serve per identificare una riga del programma assegnandole un identificatore
- Viene in genere posizionata all'inizio della riga stessa *senza indentazione* ed è terminata da un carattere ':', esempio:
  ```
  fuori:
  ```
- Tutte le label devono avere nomi diversi (stesse regole degli identificatori)
- Una label è visibile solo dall'interno della funzione dove è definita
- Una label non può essere l'ultima istruzione di un blocco (soluzione: farla seguire da un ';')

Salti incondizionati

- Un "salto" (incondizionato) fa continuare l'esecuzione di un programma dalla riga di codice identificata da una *label*
- In C si effettua per mezzo dell'istruzione `goto`:

  `goto` *label*; ← *label senza il carattere* ':'
- Una label può essere collocata in una riga precedente o successiva quella della `goto`, purché nella stessa funzione (è visibile in tutta la funzione, si dice che ha *scope di funzione*)
- Una label può essere usata da più `goto`
- Non si può saltare a una label in un'altra funz.

Salti incondizionati

- I vecchi linguaggi di programmazione non disponevano di *costrutti strutturati* e l'uso del `goto` era indispensabile
- I principali problemi ascritti all'istruzione `goto` sono i seguenti:
  - bastano pochi salti per rendere il codice intricato e il controllo del flusso difficile da seguire ("*spaghetti code*")
  - è possibile far eseguire un blocco non dall'inizio ma da più punti intermedi (*entry point*), ad esempio per eseguire solo una parte di quel blocco e non doverlo riscrivere (ma ci sono modi migliori, ad esempio usando uno `switch`)

Salti incondizionati

- Nei seguenti casi il `goto` può essere utile per questioni di efficienza e chiarezza:
  - per uscire immediatamente da cicli annidati
  - per uscire da un ciclo contenente uno `switch` (dove il `break` farebbe uscire solo dallo `switch`)
- *Si eviti l'istruzione* `goto` *in tutti gli altri casi*
- Vi sono altrettanto autorevoli sostenitori dell'uso (oculato) o del divieto assoluto d'uso del `goto` (Dijkstra contrario, Knuth a favore)
- Linguaggi più recenti (ad es. Java) non hanno `goto`, ma hanno costrutti aggiuntivi per uscire da cicli annidati (es. `break` *con etichetta*)

Salti incondizionati

- Quando si usa una `goto` per uscire da cicli annidati, si consiglia (suggerimento personale) di collocare la label *subito sotto* il corpo del ciclo più esterno (senza istruzioni intermedie) e *allineata verticalmente* con la keyword del ciclo più esterno da cui uscire

```
for (…                ← ciclo più esterno
{
    for (…            ← ciclo più interno
    { … if (condizione_speciale)
              goto fuori;
}   }
fuori:
```

Esercizi

1. Scrivere un programma che calcoli la media (con parte frazionaria) di 100 valori interi introdotti dalla tastiera.
2. Scrivere un programma che chieda quanti valori verranno introdotti dalla tastiera, li chieda tutti e ne stampi la somma e la media.
3. Scrivere un programma che calcoli la media di tutti i valori introdotti dalla tastiera finché non ne viene introdotto uno non compreso tra 18 e 30 (ad esempio 9999, provare proprio questo valore). La visualizzazione della media deve avvenire solo alla fine (ossia non ogni volta che un valore viene introdotto).

Esercizi

4. Scrivere un programma che richieda N numeri da tastiera e ne calcoli il valore massimo (leggere attentamente la nota alla soluzione).
5. Scrivere un programma che richieda N numeri da tastiera e ne calcoli il valore massimo, il valore minimo, la somma e la media.
6. Si scriva un programma che calcoli il fattoriale di un numero intero N dato dalla tastiera. Si ricordi che il fattoriale di un numero n (simbolo $n!$) viene calcolato con la seguente formula:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1.$$

Esercizi

7. Scrivere un programma che calcoli *i primi N numeri* di Fibonacci, con N introdotto dalla tastiera. I numeri di Fibonacci sono una sequenza di valori interi che inizia con i due valori fissi 1 e 1 e ogni successivo valore è la somma dei due precedenti.
   Ad esempio i primi 10 numeri di Fibonacci sono: 1 1 2 3 5 8 13 21 34 55.
8. Scrivere un programma che calcoli *i primi numeri* di Fibonacci *minori o uguali a N*, con N introdotto dalla tastiera.
   Ad esempio i primi numeri di Fibonacci minori o uguali a 10 sono: 1 1 2 3 5 8.

Esercizi

9. Scrivere un programma che verifichi se un numero x è primo. Un numero è primo se ha come divisori solo 1 e se stesso, ossia diviso per tutti i valori (interi!) intermedi dà sempre resto diverso da 0.
10. Rendere il programma precedente più efficiente considerando che non serve valutare tutti i valori intermedi, ma solo 2 e i successivi dispari fino a $\sqrt{x}$ (eventualmente incluso)
11. Scrivere un programma che calcoli i primi N numeri primi.

Esercizi

12. Si scriva un programma per calcolare ex mediante il suo sviluppo in serie:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Ogni frazione aggiunge precisione al risultato, per cui conviene usare valori di n adeguatamente elevati, ad esempio compresi tra 30 e 40. Si verifichi che i risultati calcolati in questo modo siano coerenti con quelli forniti dalla funzione intrinseca `exp` calcolando la differenza dei valori.

Esercizi

13. Si scriva un programma dove il calcolatore determini casualmente un numero intero compreso tra 0 e 99 e chieda all’utente di trovare il numero stesso. Ad ogni input dell’utente il calcolatore risponde con “troppo alto” o “troppo basso”, finché non viene trovato il valore corretto. Per generare valori casuali si utilizza la funzione `rand`.
14. Si scriva un programma per calcolare la radice quadrata mediante la formula iterativa di Newton:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{A}{x_i}\right)$$

Esercizi

(*Continuazione*)

Dato il valore A, se ne vuole calcolare la radice quadrata x. La formula data calcola valori di x sempre più precisi.

Inizialmente si considera $x_{i=0} = A$, ricavando un valore x_1 che approssima molto grossolanamente il valore della radice quadrata.

Si inserisce nuovamente x_1 nella formula (al posto di x_i) ottenendo un x_2 che è un'approssimazione migliore della precedente.

Si continua in questo modo fintanto che il risultato varia (cioè $x_{i-1} - x_i >$ `DLB_EPSILON`).