7

LE ISTRUZIONI DI CONTROLLO

Questo capitolo si occupa delle istruzioni che possono controllare il flusso del programma e l'allocazione di dati locali. Lo scopo è completare la discussione dei paragrafi 3.1 e 3.2, ove abbiamo visto l'uso delle istruzioni `if`, `if ... else` e `for`. Non discuterò ulteriormente l'uso di queste ultime istruzioni. Mi soffermerò invece sulla costruzione di blocchi e su altri modi per realizzare cicli.

7.1 Blocchi di istruzioni

Un blocco di istruzioni è formato da tutte le linee di programma che sono comprese entro una coppia di parentesi graffe, aperta e chiusa. Ad esempio, il corpo di una qualsiasi funzione costituisce un blocco di istruzioni. Di blocchi abbiamo già parlato anche a proposito delle istruzioni `if` e `for`.

La formazione di un blocco serve principalmente per isolare un gruppo di istruzioni che costituiscono una parte ben definita del programma: i cicli di iterazioni ne costituiscono l'esempio più immediato, ma non l'unico. La struttura generale di un blocco è la seguente:

`{`
  ⟨*dichiarazioni di tipo*⟩
  ⟨*istruzioni*⟩
`}`

I blocchi possono essere annidati uno entro l'altro: la sola regola da rispettare rigorosamente è che ciascun blocco deve chiudersi prima della chiusura di quello che lo contiene. È consuetudine – ben giustificata – mettere in evidenza i blocchi di programma allineando in verticale tutte le istruzioni che appartengono allo stesso blocco, ed usando il rientro per mettere in evidenza il livello di annidamento di ciscun blocco rispetto ai precedenti. I testi dei sorgenti C che trovi in queste note rispettano la convenzione.

Avrai osservato che un blocco può iniziare con delle dichiarazioni di tipo, quelle che allocano memoria locale nello stack di programma. Attenzione però a come le usi: una variabile dichiarata all'inizio di un blocco vive per tutta la durata del blocco, poi viene eliminata. Inoltre il compilatore mantiene delle tabelle locali per ciascun blocco che viene creato, organizzandole in modo gerarchico, e le usa partendo da quella del blocco più interno.

Probabilmente questa presentazione sintetica è riuscita solo a metterti in testa tanti punti interrogativi. Cerco di spiegarmi con un esempio, ma presta ben attenzione: nonostante l'apparenza non è affatto banale. Supponi di aver scritto un frammento di programma come questo:

```
{
  int j,k,m;
  j = m = 17;
  k = 13;
  {
    int j;
    j = m = k;
  }
  ...
}
```

La domanda buona è: quando il programma arriva ai puntini che valore hanno `j`, `k` e `m`? Risposta: `j` vale 17, `k` vale 13 e `m` vale 13. Sorpreso? Prova a completare il programma con qualche scrittura prima e dopo il blocco e ad eseguirlo, in modo da verificare che quel che ti ho detto è proprio vero. Poi continua a leggere: ecco la spiegazione dettagliata di quello che succede.

- All'inizio del primo blocco il compilatore crea una tabella dei nomi locali – corrispondente al primo blocco – e vi inserisce i nomi `j`, `k` ed `m` ai quali ha associato delle celle di memoria. Le due istruzioni successive assegnano il valore 17 a `j` e `m`, ed il valore 13 a `k`. Fin qui nulla di strano.
- All'ingresso del secondo blocco il compilatore crea una seconda tabella di nomi locali – corrispondente al secondo blocco – e vi inserisce il nome `j` al quale associa una cella di memoria *diversa da quella del primo blocco*.
- Qui viene l'istruzione `j=m=k`. Il compilatore cerca il nome `k` nella tabella del secondo blocco; non trovandolo, risale alla tabella del primo blocco, e qui lo trova, e ne preleva il valore che in questo momento è 13. Poi cerca il nome `m`; trova anche questo nel primo blocco, e deposita il valore 13 (prelevato da `k`) nella cella di memoria corrispondente. Infine il compilatore cerca il nome `j`, e lo trova nella tabella del secondo blocco; trovatolo, deposita il valore 13 nella cella di memoria corrispondente. Il punto cruciale è che la cella associata a `m` nel secondo blocco è *la stessa del primo*; quella associata a `j` è *diversa*.
- Alla chiusura del secondo blocco il compilatore rimuove la seconda tabella. Da questo momento il nome `j` torna ad essere associato alla cella che gli era stata assegnata nel primo blocco, che non ha subito nessuna modifica all'interno del secondo blocco. Invece il valore di `m` è

stato modificato.

Hai già visto una faccenda di questo genere quando abbiamo discusso l'uso di variabili globali e locali nelle funzioni. Ora è il momento di esporti la regola generale. Il compilatore mantiene una serie di tabelle:

- non appena inizia la lettura del file sorgente crea la tabella globale, che verrà cancellata solo alla fine del file;
- per ciascun nuovo blocco che incontra crea una tabella locale associata al blocco, che verrà immediatamente eliminata all'uscita dal blocco.

Quando deve cercare la corrispondenza tra un nome ed una cella di memoria il compilatore esamina per prima l'ultima tabella che ha creato; se non trova il nome passa alla precedente, e prosegue così fin che ha trovato il nome, oppure ha esaurito tutte le tabelle locali. In quest'ultimo caso passa a cercare il nome nella tabella globale. Se non lo trova neppure in quella segnala l'errore e se ne lava le mani — oppure assume che il nome corrisponda ad una variabile di tipo `int` che alloca al momento.

Inutile sottolineare che si tratta di un meccanismo comodo – l'uso dei blocchi può evitare l'eccessiva proliferazione di nomi e di variabili locali – ma anche delicato. Come sempre, occorre fare attenzione. Molta!

7.2 Esecuzione selettiva di blocchi

È frequente dover scrivere programmi che eseguono azioni ben diverse in corrispondenza a diverse situazioni — il caso più banale è il valore di una particolare variabile. Situazioni del genere possono ben essere gestite mediante una successione di istruzioni `if... else if ... else`. Un altro modo altrettanto efficace, con il vantaggio di una maggiore facilità di lettura, consiste nel far uso dell'istruzione `switch`.

Procediamo al solito con un esempio, che svilupperemo usando ambedue i tipi di istruzioni. Supponiamo di avere una stringa di caratteri contente delle cifre. ad esempio "Anno 1492", e di voler scrivere per esteso le cifre, "uno" al posto della cifra 1, "quattro" al posto della cifra 4, &c. [1]

7.2.1 Il solito problema: un algoritmo

Lasciami essere pedante (ma così si deve procedere quando si ha a che fare con un calcolatore). Il programma deve:

---

[1] Non lasciarti ingannare dall'apparente stupidità dell'esempio. Anzitutto, il mio scopo è illustrarti un metodo che ti serva anche in casi più utili. In secondo luogo, l'esempio è meno stupido di quanto sembri. Immagina di dover preparare una funzione che scriva per esteso la cifra su un assegno. Questo esempio è solo il punto di partenza. Per inciso, qualunque programmatore che abbia un minimo di esperienza sarebbe in grado di trovare almeno un paio di soluzioni molto più eleganti di quelle che ti sto illustrando. Ma, insisto, un esempio è un esempio.

- leggere una riga dal terminale come stringa di caratteri; la riga potrà contenere anche dei numeri;
- riscrivere la stessa riga avendo cura di sostituire le cifre con il loro nome. Ad esempio, la riga

  `Anno 1492`

  deve diventare

  `Anno uno quattro nove due`

Troppo vago, detto in questi termini. Leggere una riga di testo ormai lo dovresti saper fare: basta chiamare la funzione `scanf` con un formato di lettura `%s` e passarle come argomento l'indirizzo di un vettore `char` di lunghezza sufficiente. Ma come faccio a fare il resto?

Concentrati su quest'ultima domanda. Ecco l'algoritmo in termini un po' più precisi:

(i) crea un ciclo con un contatore intero `j` che parte da zero;
(ii) se il carattere in posizione `j` è un `NUL` (nel senso dello zero binario) termina il ciclo;
(iii) se il carattere in posizione `j` è una cifra scrivine il nome per esteso, preceduto da uno spazio;
(iv) se non si è verificato nessuno dei casi precedenti scrivi il carattere tale e quale, senza modifiche;
(v) incrementa il contatore `j` e riprendi dal passo (ii).

Al solito, verifica la correttezza dell'algoritmo, senza fretta; poi continua la lettura. Ti mostrerò due possibili realizzazioni: una usa le istruzioni `if ... else`, che già conosci. La seconda mi servirà per illustrarti l'istruzione `switch`, che per te è nuova.

7.2.2 Realizzazione con `if ... else`

Non ti dovrebbe essere difficile farlo direttamente, scimmiottando in qualche modo il programma per il calcolo della soluzione dell'equazione di secondo grado nel paragrafo 3.2. Ad ogni modo, ecco il codice corgente in C.

```
#include <stdio.h>
int main()
{
  int j;                          /* Contatore */
  char riga[128];                 /* Riga in lettura */
  printf("-> ");
  scanf("%127[ -}]",riga);        /* Leggo la riga */
  for(j=0; riga[j]!='\000'; j++) {/* (i), (ii) */
    if(riga[j] == '0')            /* (iii) */
      printf(" zero");
    else if(riga[j] == '1')
      printf(" uno");
    else if(riga[j] == '2')
```

```
      printf(" due");
    else if(riga[j] == '3')
      printf(" tre");
    else if(riga[j] == '4')
      printf(" quattro");
    else if(riga[j] == '5')
      printf(" cinque");
    else if(riga[j] == '6')
      printf(" sei");
    else if(riga[j] == '7')
      printf(" sette");
    else if(riga[j] == '8')
      printf(" otto");
    else if(riga[j] == '9')
      printf(" nove");
    else                              /* (iv) */
      printf("%c",riga[j]);
  }                                   /* (v) */
  printf("\n");
  exit(0);
}
```

Non dovresti trovare difficoltà a leggere il programma, ma qualche commento può essere utile.

- L'istruzione

  ```
  scanf("%127[ -}]",riga);
  ```

  ha uno strano formato di lettura. Interpretazione: la funzione `scanf` deve leggere una riga terminata dal tasto <Return>, limitandone la lunghezza ad un massimo di 127 caratteri (). Quello che è contenuto tra le parentesi quadre specifica quali siano i caratteri ammessi; in questo caso gli ho detto di accettare tutti i caratteri il cui codice ASCII è compreso tra quello dello spazio e quello della parentesi graffa chiusa. Se torni a guardare la tabella del codice ASCII vedrai subito che si tratta di tutti i caratteri stampabili. Se vuoi saperne di più, guarda il manuale.
- Avrai notato che il carattere `\n` che termina la riga compare solo nell'ultima istruzione `printf`, quella che precede la linea `exit(0)`.

Prova a battere, compilare ed eseguire. Forse troverai un po' antiestetico il fatto che sullo schermo ti comparirà

```
-> Anno 1492
Anno  uno quattro nove due
```

con uno spazio di troppo prima di `uno`. Se rileggerai attentamente il programma ti renderai conto che non ha fatto altro che quel che gli ho detto io: ha fatto precedere `uno` da uno spazio, senza curarsi del fatto che già ce n'era uno. Sapresti eliminare lo spazio di troppo?

7.2.3 *Realizzazione con* `switch`

E veniamo alla seconda soluzione. Procedo, come sempre, suggerendoti il codice, e lasciando i commenti a più tardi.

```
#include <stdio.h>
int main()
{
  int j;                          /* Contatore */
  char riga[128];                 /* Riga in lettura */
  printf("-> ");
  scanf("%127[ -}]",riga);        /* Leggo la riga */
  for(j=0; riga[j]!='\000'; j++) {/* (i), (ii) */
    switch(riga[j]) {
    case '0':                     /* (iii) */
      printf(" zero");  break;
    case '1':
      printf(" uno");  break;
    case '2':
      printf(" due");  break;
    case '3':
      printf(" tre");  break;
    case '4':
      printf(" quattro");  break;
    case '5':
      printf(" cinque");  break;
    case '6':
      printf(" sei");  break;
    case '7':
      printf(" sette");  break;
    case '8':
      printf(" otto");  break;
    case '9':
      printf(" nove");  break;
    default:                     /* (iv) */
      printf("%c",riga[j]);
    }
  }                               /* (v) */
  printf("\n");
  exit(0);
}
```

Qui i commenti sono d'obbligo, e riguardano proprio l'uso dell'istruzione `switch`. Tanto vale descriverla in generale. La sintassi è:

```
switch(⟨espressione intera⟩) {
case j1:
  ⟨istruzioni⟩
case j2:
  ⟨istruzioni⟩
... ...
default:
  ⟨istruzioni⟩
}
```

Ed ecco il significato.

- Viene valutata l'⟨*espressione intera*⟩ tra parentesi; per intero si intende anche il tipo `char`.
- L'esecuzione riprende dall'etichetta `case` il cui valore corrisponde a quello calcolato. Il valore specificato dopo `case` deve essere una costante: non sono ammesse variabili né espressioni.
- L'esecuzione continua fin che si incontra l'istruzione `break`, oppure fino alla parentesi graffa che chiude il blocco `switch`. La presenza di etichette non modifica per nulla l'esecuzione: l'etichetta costituisce un punto di riferimento per un salto, ma non provoca salti né interruzioni. In altre parole, se non c'è un `break` l'esecuzione prosegue tranquillamente. Presta attenzione a quest'ultimo punto: non è infrequente dimenticare il `break` e poi scoprire che il programma esegue istruzioni non volute.
- L'etichetta `default` copre tutti i casi non esplicitamente specificati. Non è obbligatoria: se viene omessa, tutti i casi non esplicitamente previsti sono ignorati.

Nel nostro esempio sono esplicitamente previsti i casi in cui il carattere ha il valore ASCII che rappresenta una cifra numerica; tutti gli altri sono classificati cone `default`. Ogni caso si chiude con un `break`.

Non è proibito raggruppare più etichette. Ad esempio, le istruzioni

```
  switch(riga[j]) {
  case '0':
  case '2':
  case '4':
  case '6':
  case '8':
    printf(" pari");
  case '1':
  case '3':
  case '5':
  case '7':
  case '9':
    printf(" dispari");
}
```

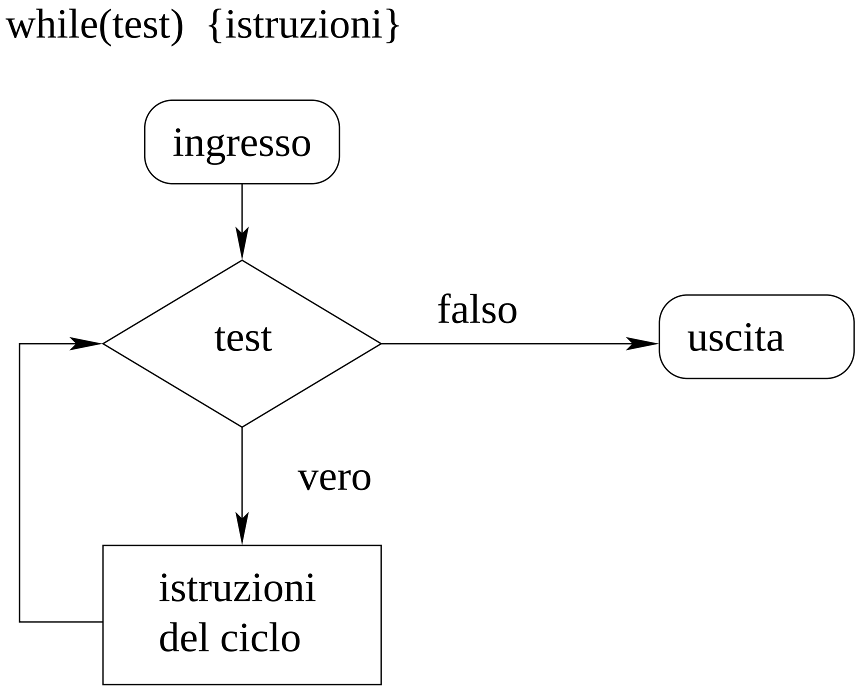

Figura 7.1. Il ciclo iterativo realizzato con l'istruzione `while`. Il test viene eseguito immediatamente all'ingresso del ciclo.

potrebbero servire per scrivere semplicemente `pari` o `dispari` al posto della cifra. Non essendoci un caso `default` ogni altro carattere verrebbe ignorato. Nel dubbio, prova!

7.3 Cicli di istruzioni

Abbiamo già visto come realizzare un ciclo iterativo sia, in modo un po' artigianale, con l'accoppiata di istruzioni `if` e `goto`, sia con l'istruzione `for`. In linea di massima questo è già sufficiente per scrivere un gran numero di programmi. Tutto quello che sto per raccontarti è un utile complemento, ma non strettamente indispensabile. Le istruzioni sono `while` e `do...while`. Le puoi considerare come delle versioni semplificate dell'istruzione `for`.

7.3.1 Il ciclo `while`

La sintassi è:

`while(`⟨*espressione logica*⟩`) {`⟨*istruzioni*⟩`}`

Il diagramma di flusso è rappresentato in figura 7.1. All'ingresso del ciclo viene immediatamente eseguito il test di controllo; se il risultato è falso le istruzioni del ciclo vengono semplicemente ignorate. Altrimenti il ciclo viene ripetuto fin che il test dà come risultato vero. Naturalmente, le istruzioni del ciclo dovranno contenere qualche operazione che possa modificare il risultato del test, altrimenti il ciclo rischia di continuare all'infinito: questa situazione, particolarmente temuta dai programmatori dell'epoca in cui l'uso del calcolatore doveva essere pagato in moneta sonante, viene comunemente denominata *loop*.

Come esempio di ciclo `while` consideriamo la situazione seguente: voglio sapere quanti caratteri contiene una stringa. Per questo mi basta ricordare che la convenzione del linguaggio C è che ogni stringa debba essere terminata da un carattere `NUL`, lo zero binario. Se rifletti un momento, ti renderai subito conto che la lunghezza della stringa è data proprio dalla posizione del primo `NUL`. Scrivere l'algoritmo per questo frammento di programma è facile:

(i) inizializza un contatore `j` a zero;
(ii) incrementa il contatore `j` fin che il carattere nella posizione `j` non è `NUL`.

La scrittura dell'algoritmo in C può realizzarsi mediante il ciclo `while`. Ecco il frammento di programma, dove suppongo che `j` sia stato dichiarato `int`, e `riga` sia il vettore che contiene la stringa di caratteri:

```
j=0; while(riga[j] != '\000') ++j;
```

All'uscita dal ciclo il contatore `j` contiene la lunghezza della stringa. Nota che ho omesso le parentesi graffe: il ciclo è costituito da un'unica istruzione.

Naturalmente, lo stesso risultato si può ottenere anche con un ciclo `for`. Eccoti l'istruzione:

```
for(j=0; riga[j]!='\000', j++);
```

Quale delle due è migliore? Risposta: quella che ti piace di più. Se provi ad immaginare come avverrà l'esecuzione ti renderai conto che le operazioni eseguite sono esattamente le stesse.

7.3.2 Il ciclo `do ... while`

La sintassi è:

do {⟨*istruzioni*⟩} while(⟨*test*⟩)

Il diagramma di flusso è rappresentato in figura 7.2. Rispetto al ciclo `while` presenta una sola differenza: il test per decidere se ripetere il ciclo viene eseguito dopo le istruzioni, e non prima. Di conseguenza, le istruzioni del ciclo vengono eseguite almeno una volta. Quest'ultima circostanza costituisce tipicamente il vero motivo della scelta tra la codifica di un ciclo con `while` o con `do ... while`. Facciamo un paio di esempi.

Supponiamo di voler programmare la stessa operazione che abbiamo fatto poco fa – determinare la lunghezza di una stringa di caratteri – mediante `do ... while`. Dovremmo scrivere, ad esempio

```
j= -1;  do {++j}  while(riga[j] != '\000');
```

Un po' innaturale, quel contatore che parte da `-1`, ma inevitabile proprio perché l'istruzione del ciclo viene eseguita almeno una volta. Punto a favore: funziona.

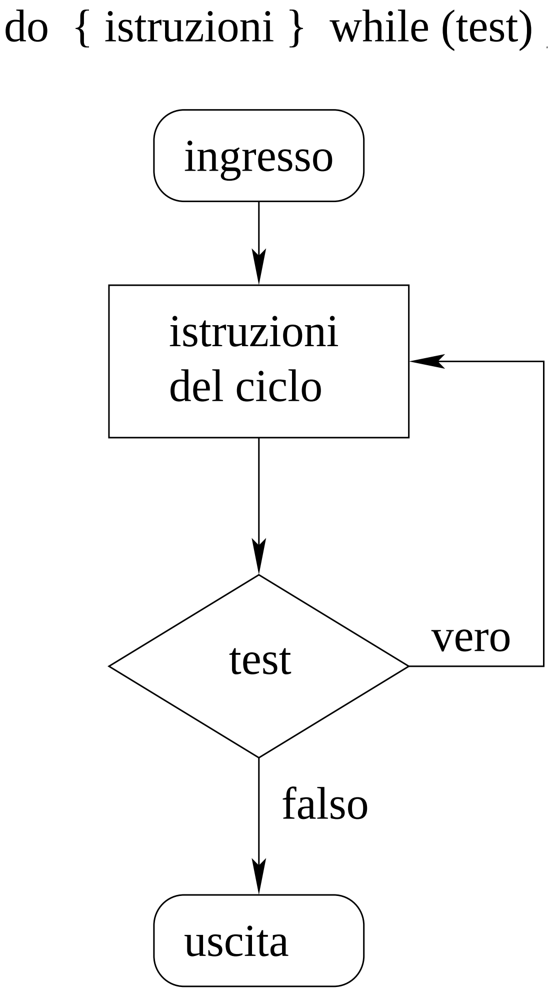

Figura 7.2. Il ciclo iterativo realizzato con l'istruzione `do ... while`. Il test viene eseguito alla fine delle istruzioni del ciclo; di conseguenza le istruzioni vengono eseguite almeno una volta.

Un caso in cui l'uso del `do ... while` risulta più naturale è il seguente. Supponiamo di voler stampare in binario l'intero contenuto di un dato `short int`, compresi gli eventuali bit nulli che ci sono in testa. Forse il problema non ti sembrerà banale, quindi inizio a darti qualche suggerimento.

Il trucco sta nell'usare le operazioni bit–a–bit, incluso lo shift. Se io ti scrivessi una sequenza di cifre binarie, ad esempio `0110010110110110` e ti chiedessi di leggermela tu punteresti il dito sulla prima cifra, e leggeresti "zero", poi sulla seconda e leggeresti "uno" &c, fino all'esaurimento delle cifre. Questo è esattamente quanto puoi fare usando le istruzioni bit–a–bit. Il dito è costituito da quella che si chiama una *bitmask*, o *maschera di bit*: un dato intero che inizialmente contiene la configurazione `1000...0` — la lunghezza dipende dal tipo di dato. Se ne faccio l'operazione di *and* bit–a–bit con un dato intero `k` della stessa lunghezza il risultato sarà divero da zero se e solo se il bit più significativo di `k` è `1`. Poi sposterò a destra il bit della maschera con uno *shift*; otterrò `0100...0`, e l'operazione di *and* bit–a–bit mi dirà se il secondo bit da destra è `1` o `0`, &c. Il procedimento terminerà quando lo shift verso destra mi avrà completamente azzerato la maschera di bit.

Riscriviamo la stessa cosa in un algoritmo ben ordinato.

(i) Crea una maschera di bit con `100...00`;

(ii) Esegui un *and* bit–a–bit della maschera col dato;

  (ii.a) se il risultato è zero scrivi `0`;

  (iii.b) altrimenti scrivi `1`

(iii) esegui uno shift a destra della maschera di bit;

(iv) se la maschera non è zero riprendi dal passo (ii), altrimenti

(v) la scrittura è terminata.

Ora vediamo il codice C del frammento di programma, supponendo che il dato sia di tipo `short int`. Qui, `k` è il dato che voglio scrivere in binario, e `mask` è la maschera di bit.

```
unsigned short int k,mask;
  ... definisci k ...
mask = 0x8000;
do {
  if((mask & k) == 0) printf("0");
  else printf("1");
} while((mask >>= 1) != 0);
```

E, infine, qualche commento.

- Ho definito la maschera di bit `mask` come `unsigned` per evitare il prolungamento del segno durante lo shift a destra: sarebbe il *loop* assicurato!
- Per assegnare alla maschera il valore iniziale il modo più comodo è far ricorso alla rappresentazione esadecimale. So che un dato `short int` ha una lunghezza di 16 bit — sulla mia macchina. Quindi devo memorizzare in `mask` la sequenza di bit `1000 0000 0000 0000`. Raggruppando i bit a quattro a quattro e andando a rivedere la tabella di conversione nella tavola 4.3 trovo che in esadecimale devo scrivere `0x8000`; ciò che ho fatto.

Il resto altro non è che la realizzazione dell'algoritmo che ti ho esposto. Non ti sarà difficile completare le istruzioni inserendo ad esempio la lettura di `k`, e tutto quello che serve per avere un testo sorgente completo. Prova!

Naturalmente, non è proibito eseguire la stessa operazione usando un ciclo `while`. Ecco cosa cambierebbe:

```
mask = 0x8000;
while(mask != 0) {
  if((mask & k) == 0) printf("0");
  else printf("1");
  mask >>= 1;
}
```

Tutto identico? Non esattamente: questo ciclo esegue il test anche sul valore iniziale di `mask`, il che costituisce un'operazione inutile. Sembra un'inezia, ed in questo caso in fondo lo è. Ma non sottovalutare l'osservazione. Le istruzioni inutili aumentano il tempo di esecuzione; se queste facessero parte di una funzione che viene richiamata continuamente la perdita di tempo si accumulerebbe, e potrebbe diventare consistente.

7.3.3 L'interruzione di un ciclo

Ci sono due istruzioni che consentono l'interruzione di un ciclo realizzato con una qualunque delle istruzioni `for`, `while` o `do ... while`. Le istruzioni sono `break` e `continue`.[2] Ambedue interrompono il flusso sequenziale dell'esecuzione; la differenza è la seguente:

- `break` interrompe completamente il ciclo iterativo, saltando alla prima istruzione *dopo* il ciclo;
- `continue` termina l'iterazione corrente e passa alla successiva.

Come esempio di uso dell'istruzione `break` supponiamo di voler scrivere una funzione che determini se una stringa contenga o no un determinato carattere, ed in caso affermativo ne restituisca la posizione all'interno della stringa stessa.

Chiameremo la funzione `cerca_carattere`; gli argomenti saranno un vettore `str` di tipo `char` che contiene la stringa, ed un dato `c` anch'esso di tipo `char` che specifica quale sia il carattere da cercare; il valore restituito dovrà essere un intero che rappresenti la posizione del carattere cercato all'interno della stringa, oppure la lunghezza della stringa stessa.

L'algoritmo è semplice: basta realizzare un ciclo che fa scorrere tutti i caratteri della stringa, e viene interrotto se si trova il carattere cercato; in caso di interruzione l'indice di controllo del ciclo dà direttamente la posizione del carattere. Il ciclo dovrà comunque interrompersi alla fine della stringa, identificata come sempre da un carattere `NUL`.

Ecco il codice C della funzione:

```
int cerca_carattere(str,c)
     char str[],c;
```

---

[2] Abbiamo già incontrato l'istruzione `break` in coppia con `case`. Il significato qui è simile: il blocco corrente di istruzioni viene interrotto. L'istruzione `continue` non può essere usata entro un blocco `switch`, perché in quel caso non c'è nessun ciclo in atto.

```
{
  int j;
  for(j=0; str[j]!='\000'; j++)
    if(str[j]==c) break;
  return(j);
}
```

Questa volta non dovresti aver bisogno di altre spiegazioni. Può darsi però che ti venga spontanea una domanda: ma l'istruzione `break` è equivalente ad un `goto` alla prima istruzione dopo il ciclo? In termini più precisi, la domanda è se non si possa scrivere il ciclo della funzione `cerca_carattere` anche nella forma

```
for(j=0; str[j]!='\000'; j++) {
  if(str[j]==c) goto trovato;
}
trovato:
...
```

Così è, in effetti. L'uso di `break` invece che di un `goto` risparmia un'etichetta, ed anche la fatica di cercare dove si trova: il salto è alla fine del ciclo.

L'istruzione `continue` interrompe solo l'iterazione corrente del ciclo, e passa alla successiva. In altre parole, è equivalente ad un `goto` alla parentesi graffa che chiude il corpo del ciclo. Ad esempio, immagina di dover scrivere una funzione che riceva come argomento una stringa di caratteri, e debba eseguire una serie di operazioni complesse quando incontra una cifra, ignorando tutti gli altri caratteri. La funzione potrà contenere un ciclo così costruito:

```
for(j=0; str[j]!='\000'; j++) {
  if(str[j]<'0' && str[j]>'9') continue;
  ⟨istruzioni per gestire le cifre⟩
}
```

Osserverai che avresti potuto ottenere lo stesso scopo con `if...else`. È perfettamente vero; l'uso di `continue` può essere utile al fine di ridurre il numero di livelli di annidamento dei blocchi di istruzioni, ma non è essenziale.