Capitolo 3

Costrutti per il controllo di sequenza

Nella descrizione di un algoritmo è fondamentale poter specificare l'ordine in cui si susseguono le diverse istruzioni ovvero, il *controllo* del flusso d'esecuzione (o *controllo di sequenza*). Nei diagrammi di flusso, abbiamo visto, l'ordine è specificato molto semplicemente tramite frecce che connettono le diverse istruzioni. Nei linguaggi di basso livello, tipicamente, sono presenti a questo scopo *istruzioni di salto* che permettono di modificare, quando opportuno, il comportamento di "default" della macchina hardware che è quello di eseguire le singole istruzioni una dopo l'altra, in sequenza, così come appaiono memorizzate nella memoria principale del calcolatore. I linguaggi di programmazione ad alto livello, di tipo convenzionale, offrono un più ricco insieme di costrutti sintattici per specificare, in termini più astratti, diverse tipiche forme (o *strutture*) di controllo di sequenza. In tutti questi linguaggi sono presenti statement per specificare (almeno) le seguenti strutture di controllo:

- sequenzializzazione
- selezione
- iterazione
- salto.

Per quanto riguarda la *sequenzializzazione* i linguaggi ad alto livello adottano solitamente soluzioni molto semplici come, ad esempio, l'uso esplicito di un operatore di sequenzializzazione (ad esempio il **;** in Pascal) o l'assunzione che la sequenzializzazione sia la regola di default. Ad esempio, in C++ gli statement sono eseguiti in sequenza, uno dopo l'altro, come appaiono nel testo, a meno che non si trovino all'interno di uno statement che modifica esplicitamente il flusso d'esecuzione.

Nota. In C++ il **;** è semplicemente un "terminatore" di statement (e di dichiarazioni), e non un operatore che indica la sequenzializzazione di due statement, come accade ad

esempio in Pascal. Per questo in C++, qualsiasi statement (o dichiarazione, tranne lo statement composto) deve essere terminata da un `;`, indipendentemente da ciò che segue (ad esempio, in C++, l'ultimo statement del programma, prima della parentesi graffa finale, va comunque terminato da un `;`).

Nei paragrafi successivi descriveremo in dettaglio gli statement offerti dal C++ per modellare le diverse strutture di controllo sopra citate. In particolare analizzeremo:

- gli statement `if` e `switch` per le strutture di controllo selettive;
- gli statement `while`, `do-while` e `for` per le strutture di controllo iterative;
- gli statement `break`, `continue` e `goto` per i salti espliciti.

Altri costrutti e meccanismi che permettono di realizzare altre forme di astrazione sul controllo, quali quelli per la definizione e la chiamata di sottoprogrammi ed il meccanismo della ricorsione, verranno presi in esame in un capitolo successivo (Capitolo 5).

Prima di descrivere in dettaglio i singoli statement, introduciamo la nozione di *statement composto* che utilizzeremo poi nella presentazione degli statement stessi.

Nota. Si osservi che le *espressioni*, introdotte nel capitolo precedente, costituiscono già una forma di controllo del flusso d'esecuzione. Nella valutazione di un'espressione, infatti, si procede seguendo un ben preciso ordine tre le varie parti costituenti l'espressione: tipicamente, valutando prima gli operandi, da sinistra a destra, e poi applicando l'operatore e, nel caso in cui un'espressione sia composta da più operatori, seguendo precise regole di precedenza ed associatività.

3.1 Statement composto

I linguaggi di programmazione convenzionali offrono normalmente un costrutto—detto *statement composto*—che permette di raggruppare più statement per formare un unico statement. Lo statement composto in C++ ha la seguente forma:.

$$\begin{array}{l}\{\ \textit{dichiarazioni} \\ \qquad \cup \\ \quad \textit{statement} \\ \}\end{array}$$

e cioè un insieme di dichiarazioni (eventualmente vuoto) unito ad un insieme di statement (eventualmente vuoto), il tutto racchiuso tra una coppia di parentesi graffe.

Esempio 3.1 (Statement composto) *Il seguente frammento di codice C++ costituisce un esempio di statement composto:*

```
{ int x;
  cin >> x;
  float y;
  y = x/2.0;
  cout << y;
}
```

La regione di testo racchiusa tra le due graffe costituisce un *blocco*, nozione fondamentale per la strutturazione dei programmi nei linguaggi di programmazione moderni che incontreremo nuovamente e preciseremo nel sottocapitolo 3.10 dedicato ai problemi di visibilità degli identificatori. Per ora anticipiamo soltanto che tutti gli identificatori eventualmente dichiarati all'interno di un blocco sono *locali* al blocco e cioè utilizzabili soltanto all'interno di quel blocco o di eventuali altri blocchi in esso contenuti.

Si osservi che un programma principale, come quello mostrato nel sottocapitolo 1.4.2, è costituito da una *testata*, che di base ha la forma `int main()`, seguita dal *corpo* del programma che di fatto è uno statement composto.

3.2 Statement `if`

Tra le strutture di controllo selettive più comuni figurano senz'altro quelle che permettono l'"esecuzione condizionale" di un'istruzione e la "biforcazione", già incontrate nel sottocapitolo 1.2 relativo ai diagrammi di flusso. Queste strutture di controllo sono realizzate nella maggior parte dei linguaggi di programmazione attuali da una o più forme di statement `if`. Vediamo sintassi e semantica di questo statement nel caso del linguaggio C++.

3.2.1 Caso base

Sintassi

`if (`E`)` S`;`

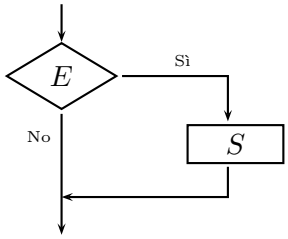
dove

E: espressione booleana;
S: statement qualsiasi.

Semantica (informale)

Calcola il valore dell'espressione E; se E ha valore "vero" allora esegue lo statement S e quindi termina l'esecuzione dello statement; altrimenti, termina immediatamente.

Il caso descritto è rappresentabile con un diagramma di flusso nel seguente modo (*selezione singola*):



Si tratta di una forma di *esecuzione condizionale*: lo statement S viene eseguito solo se la condizione E è vera.

Esempio 3.2 *Il seguente semplice frammento di programma calcola il valore assoluto di un numero intero letto da standard input:*

```
int x;
cin >> x;
if (x < 0)
  x = -x;
cout << "Il valore assoluto e' " << x;
```

Lo statement `x = -x` *viene eseguito soltanto se la condizione* `x < 0` *è vera. In ogni caso, terminata l'esecuzione dell'*`if`*, si passa allo statement successivo, che, in questo esempio, provvederà a stampare il valore corrente della variabile* `x`*.*

Qualora le istruzioni da eseguire quando la condizione dell'`if` è vera siano più di una, è necessario racchiuderle all'interno di uno statement composto, in modo che possano essere viste come un unico statement (si osservi che la sintassi dell'`if` prevede che S sia un singolo statement, che potrebbe comunque essere anche lo statement composto). Come esempio, si supponga di voler modificare il frammento di programma mostrato nell'Esempio 3.2 in modo che, quando `x < 0` è vero, allora venga stampato anche un messaggio che informa che il numero dato è negativo. Lo statement `if` dell'esempio 3.2 viene sostituito da:

```
if (x < 0) {
  cout << "Il numero dato e' negativo" << endl;
  x = -x;
}
```

Nota. Si osservi che senza racchiudere tra graffe i due statement da eseguire quando la condizione dell'`if` è vera il programma risultante

```
int x;
cin >> x;
if (x < 0)
  cout << "Il numero dato e' negativo" << endl;
  x = -x;
cout << "Il valore assoluto e' " << x;
```

avrebbe, in generale, un comportamento totalmente diverso (e non corretto rispetto alle nostre intenzioni originarie). Infatti, in questo caso, il compilatore considera come parte del costrutto `if` l'istruzione di stampa `cout << ... << endl`, ma non l'istruzione di assegnamento successiva, che pertanto è eseguita sempre, indipendentemente dal valore dell'espressione `x < 0`. Dunque, se, ad esempio, il dato in input è `-3`, il programma stampa correttamente

```
Il numero dato e' negativo
Il valore assoluto e' 3
```

ma, se il dato in input è `3`, il programma stampa

```
Il valore assoluto e' -3
```

che evidentemente non è il risultato desiderato.

3.2.2 Caso `if-else`

Lo statement `if` ammette anche un'altra forma, in cui è prevista una parte introdotta dalla parola chiave `else`.

Sintassi

`if (`E`)` S_1`;`
`else` S_2`;`

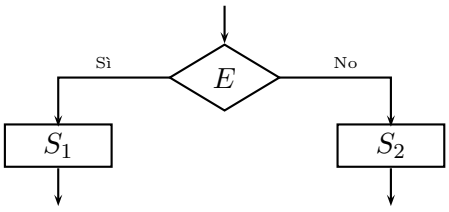
dove

E: espressione booleana;
S_1, S_2: statement qualsiasi.

Semantica (informale)

Calcola il valore dell'espressione E; se E ha valore "vero" allora esegue lo statement S_1, altrimenti esegue lo statement S_2; quindi, in entrambi i casi, termina l'esecuzione dello statement.

Anche in questo caso la situazione è rappresentabile con un diagramma di flusso (*selezione doppia*):



Si ha dunque una *biforcazione* nel flusso di esecuzione: si esegue uno tra gli statement S_1 o S_2 in base al verificarsi o meno della condizione E.

Esempio 3.3 *Il seguente semplice frammento di programma determina il maggiore tra due numeri interi letti da standard input e lo scrive sullo standard output:*

```
int x, y, max;
cin >> x >> y;
if (x < y)
  max = x;
else
  max = y;
cout << "Il maggiore e' " << max;
```

(per il diagramma di flusso relativo allo statement `if-else` *di questo esempio si veda il sottocapitolo 1.2).*

Anche per lo statement che compare nella parte `else` valgono le stesse considerazioni fatte nel caso dell'`if` semplice: qualora le istruzioni da eseguire siano più di una è necessario che siano racchiuse tra le parentesi graffe di uno statement composto, come mostra il seguente frammento di codice:

```
if (x >= y) {
  max = x;
  min = y;
}
else {
  max = y;
  min = x;
}
```

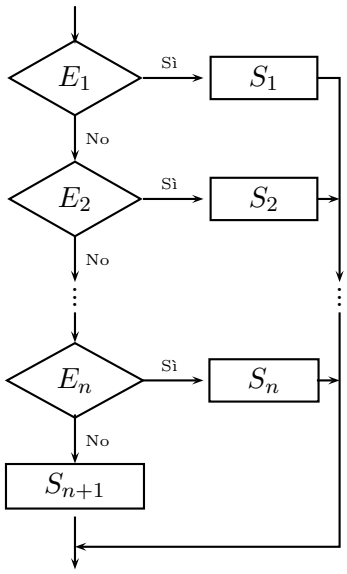
Si osservi che lo statement composto non vuole il "`;`" dopo la "`}`".

3.2.3 Statement `if-else` annidati

Gli statement S, S_1 ed S_2 delle diverse forme di `if` possono essere statement qualsiasi. In particolare possono essere a loro volta statement `if`, nel qual caso si parla di `if` "annidati" (o "nidificati"). Un caso particolarmente interessante di `if` annidati è quello che realizza una struttura di controllo del tipo *"test multiplo"* (o *selezione multipla*):

```
if (E1)  S1;
else
  if (E2)  S2;
  else
    ...
      else
        if (En)  Sn;
        else Sn+1;
```

Come mostra l'indentazione, il ramo `else` relativo al primo `if` contiene a sua volta un `if`; il ramo `else` relativo al secondo `if` contiene ancora un `if` e così via. La struttura di controllo realizzata da queste istruzioni è quella descritta dal seguente diagramma di flusso:



Si osservi che le espressioni booleane E_1, E_2, ..., vengono valutate nell'ordine in cui compaiono. Pertanto lo statement S_i eseguito sarà quello associato alla prima condizione S_i che risulta vera. In particolare, lo statement S_n viene eseguito solamente nel caso in cui tutte le precedenti espressioni booleane abbiano valore falso. In ogni caso, soltanto uno degli statement S_i viene eseguito, mentre tutti gli altri vengono ignorati.

Nota. Per evidenziare meglio la struttura del test multiplo può essere conveniente utilizzare una diversa indentatura degli statement `if` annidati, come mostrato qui di seguito:

```
if (E1)  S1;
else if (E2)  S2;
...
else if (En)  Sn;
else  Sn+1;
```

Il seguente esempio di programma completo mostra l'utilizzo di più `if` annidati per realizzare un test multiplo.

Esempio 3.4 (Conversione *voti* → *giudizi*)
Problema. *Scrivere un programma in grado di convertire un voto numerico tra 0 e 10 in un giudizio, secondo il seguente schema:*

voto $\leq$ *5,* *giudizio: insufficiente*
5 $<$ *voto* $\leq$ *6.5,* *giudizio: sufficiente*
6.5 $<$ *voto* $\leq$ *7.5,* *giudizio: buono*
voto $>$ *7.5,* *giudizio: ottimo*

Input*: un numero reale.*
Output*: una stringa di caratteri (cioè il giudizio) oppure un messaggio di errore.*

Programma*:*

```
#include <iostream>
using namespace std;
int main() {
  float voto;
  cout << "Dammi il voto numerico (tra 0 e 10)" << endl;
  cin >> voto;
  if (voto < 0 || voto > 10)
    cout << "voto non valido" << endl;
  else if (voto <= 5)
    cout << "insufficiente" << endl;
  else if (voto <= 6.5)
    cout << "sufficiente" << endl;
  else if (voto <= 7.5)
    cout << "buono" << endl;
  else
    cout << "ottimo" << endl;
  cout << "arrivederci" << endl;
  return 0;
}
```

Esempio d'esecuzione *(le parti sottolineate rappresentano input):*

```
Dammi il voto numerico (tra 0 e 10)
6
sufficiente
arrivederci
```

Nota*. Si noti che nei test relativi ai giudizi "sufficiente" e "buono" non è necessario specificare anche l'estremo inferiore del voto. Ad esempio, per il giudizio buono basta dare la condizione* `(voto <= 7.5)` *e non necessariamente* `(voto > 6.5 && voto <= 7.5)` *dato che, nel momento in cui si esegue il test, sicuramente sono già stati effettuati i test precedenti e quindi sono già stati esclusi i valori di voto inferiori o uguali a* `5`*.*

Esercizio 3.5 *Scrivere un programma C++ che legge da standard input due numeri interi e determina se i due numeri sono uguali o se uno è maggiore dell'altro e, nel primo caso, stampa il messaggio "*`I due numeri`

`sono uguali`" mentre, nel secondo caso, stampa il maggiore tra i due numeri (SUGG.: completare il frammento di programma dell'Esempio 3.3, utilizzando due `if` *annidati).*

Variante #1 per l'esempio 3.4.
Il programma dell'esempio 3.4 può essere riscritto utilizzando una sequenza di statement `if` semplici, piuttosto che più `if-else` annidati. Mostriamo qui soltanto le parti del programma modificate:

```
...
cin >> voto;
if (voto < 0 || voto > 10)
  cout << "voto non valido" << endl;
if (voto > 0 && voto <= 5)
  cout << "insufficiente" << endl;
if (voto > 5 && voto <= 6.5)
  cout << "sufficiente" << endl;
if (voto > 6.5 && voto <= 7.5)
  cout << "buono" << endl;
if (voto > 7.5 && voto <= 10)
  cout << "ottimo" << endl;
cout << "arrivederci" << endl;
...
```

I risultati prodotti dalle due versioni del programma sono gli stessi, ma la nuova versione (variante #1) risulta meno soddisfacente per almeno due motivi:

- è più pesante da scrivere, in quanto richiede di esplicitare tutte le condizioni, indicando sia estremo inferiore che superiore dei diversi intervalli;
- è più inefficiente, in quanto richiede di eseguire sempre comunque tutti i test, anche se uno soltanto avrà effettivamente esito positivo (e cioè la condizione risulterà vera).

Esercizio 3.6 *Scrivere i diagrammi di flusso delle due versioni dei programmi per la conversione di voti in giudizi e confrontarli tra loro.*

Approfondimento (Ambiguità tra `if` annidati). L'utilizzo di `if-else` annidati può portare a situazioni di possibile ambiguità. Si consideri ad esempio il seguente frammento di programma:

```
if (trovato == true)
if (a == 0)
cout << "a e' nullo" << endl;
else
cout << "a non e' nullo" << endl;
```

A quale `if` corrisponde il ramo `else`? L'ambiguità è risolta in C++—come nella maggior parte dei linguaggi convenzionali—assumendo che un `else` faccia riferimento sempre all'`if` più vicino che non abbia già un `else` associato. Quindi, nell'esempio di sopra, l'`else` si riferisce al secondo `if` (se ci fosse anche un altro `else`, successivo a quello già presente, questo ulteriore `else` farebbe chiaramente riferimento al primo `if`, in quanto il secondo sarebbe già "completo"). L'indentatura del programma può facilitare la comprensione dell'annidamento degli `if`. Conviene quindi scrivere l'esempio di sopra come:

```
if (trovato == true)
  if (a == 0)
    cout << "a e' nullo" << endl;
  else
    cout << "a non e' nullo" << endl;
```

Se vogliamo che il ramo `else` si riferisca al primo `if` è necessario inserire il secondo `if` all'interno di uno statement strutturato nel modo seguente:

```
if (trovato == true) {
  if (a == 0)
    cout << "a e' nullo" << endl;
}
else
  cout << "Non e' stato trovato l'elemento cercato" << endl;
```

3.3 Statement `while`

Un altro tipo di struttura di controllo che si incontra con grande frequenza nella progettazione di algoritmi è la struttura di controllo iterativa, caratterizzata dall'esecuzione ripetuta di una sequenza di istruzioni (un ciclo). Nella pratica, si individuano diverse tipologie di strutture iterative, distinte, ad esempio, in base alla posizione in cui compare la condizione d'uscita dal ciclo o al tipo di azioni che si compiono all'interno del ciclo. Tutti i linguaggi di programmazione convenzionali forniscono uno o più costrutti linguistici che permettono di realizzare, in modo più o meno naturale, le diverse tipologie di strutture di controllo iterative. Il C++, in particolare, offre tre diversi statement, `while`, `do-while` e `for`, specificatamente rivolti alla realizzazione di cicli, oltre alle istruzioni di salto esplicito che possono comunque essere utilizzate per la realizzazione di cicli.

In questo sottocapitolo descriveremo lo statement `while`, mentre gli altri statement iterativi e quelli di salto verranno descritti nei paragrafi successivi di questo sottocapitolo.

Sintassi

`while (`E`)`
S`;`

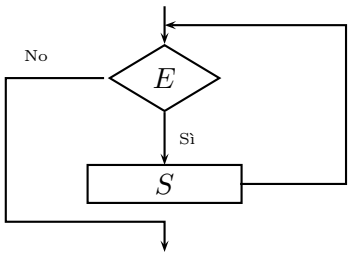
dove

E: espressione booleana;
S: statement qualsiasi.

Semantica (informale)

Calcola il valore dell'espressione E; se E ha valore "vero" allora esegue lo statement S e ripete dall'inizio; se E ha valore "falso" allora termina l'esecuzione dello statement.

La situazione è descrivibile con un diagramma di flusso nel seguente modo:



Lo statement `while` realizza in modo naturale una struttura di controllo iterativa in cui la condizione d'uscita dal ciclo è posta all'inizio ed in cui le azioni da ripetere (il "corpo del ciclo") possono essere qualsiasi. Di fatto, il ciclo continua finchè la condizione E non diventa falsa.

Ogni ripetizione dello statement S viene detta *iterazione* del ciclo. In generale, non è possibile stabilire a priori il numero di iterazioni, ma si osservi che, nel caso in cui l'espressione booleana E risulti subito falsa, allora non vi è alcuna iterazione (lo statement `S` non viene quindi eseguito, e si passa subito allo statement successivo al `while`).

Come nel caso dello statement `if`, qualora le istruzioni all'interno del ciclo siano più di una, è necessario che vengano racchiuse in uno statement composto, ossia tra parentesi graffe.

Un semplice esempio di utilizzo del `while` è mostrato nel seguente frammento di programma.

Esempio 3.7 *Stampa i quadrati dei primi 10 numeri interi positivi:*

```
int i = 1;
while (i <= 10){
   cout << i * i << endl;
   i = i + 1;
}
cout << "terminato!" << endl;
```

L'esecuzione di questi statement produrrà il seguente output:

```
1
4
9
...
100
terminato!
```

Il corpo del `while` può contenere anche altri statement strutturati, in particolare altri statement `while` (cicli annidati, di cui vedremo esempi più avanti) o statement `if`. Il seguente programma completo mostra un esempio di `while` contenente uno statement `if`.

Esempio 3.8 (Conta numero totale di vocali)
Problema. *Scrivere un programma che legge da standard input una sequenza di caratteri terminata da un punto, determina il numero di vocali minuscole presenti nella sequenza e quindi scrive il numero calcolato sullo standard output.*

Programma:

```
#include <iostream>
using namespace std;
int main() {
  char c;
  int vocali_minusc = 0;
  cout << "Inserisci sequenza di caratteri terminata da ."
       << endl;
  cin >> c;   //legge un carattere e lo assegna a c
  while (c != '.'){
    if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
       ++vocali_minusc;
   cin >> c;
  }
  cout << "La sequenza data contiene " << vocali_minusc
       << " vocali minuscole" << endl;
  return 0;
}
```

L'esecuzione del programma, con la sequenza di input

```
Una frase di prova.
```

produce il seguente output:

```
La sequenza data contiene 6 vocali minuscole.
```

Esercizio 3.9 *Scrivere un programma C++ che legge da standard input una sequenza di numeri interi terminata da un numero negativo, calcola la media aritmetica dei numeri (non negativi) letti, e scrive il risultato sullo standard output (il programma controlla anche che la sequenza non sia vuota, nel qual caso informa l'utente con opportuno messaggio in output). (SUGG.: la struttura del programma è essenzialmente la stessa del programma dell'Esempio 3.8; si utilizza una variabile* `somma` *in cui "accumulare", man mano, la somma parziale dei numeri letti da input, inserendo all'interno del ciclo uno statement* `somma = somma + x`*, dove* `x` *è la variabile che contiene l'ultimo numero letto da input, . . .).*

Esercizio 3.10 *Scrivere un programma che legge da standard input una sequenza di caratteri terminata da un punto e determina e stampa il numero di "doppie" presenti nella sequenza, dove per "doppie" si intende una sequenza di due caratteri consecutivi qualsiasi (ma diversi da spazio e a capo) identici. Ad esempio, data in input la sequenza*

`Arrivero'  appena  possibile.`

il programma deve indicare che sono presenti 3 doppie.

Nota (Cicli senza fine). Un aspetto critico delle strutture di controllo iterative, come quelle realizzate tramite il `while`, è dato dalla possibilità di scrivere cicli non terminanti. In generale, è compito di chi progetta l'algoritmo assicurarsi che i cicli presenti terminino in un tempo finito.

Un primo semplice controllo è assicurarsi che il valore dell'espressione che costituisce la condizione d'uscita dal ciclo venga in qualche modo modificata all'interno del ciclo (altrimenti la condizione, se inizialmente vera, rimarrebbe tale per sempre e quindi non porterebbe mai all'uscita dal ciclo). Si consideri, ad esempio, il seguente frammento di programma C++:

```
int i = 0;
while (i < 10)
  somma = somma + 1;
```

Poichè il valore della variabile che funge da variabile di controllo del ciclo (in questo caso la variabile `i`) non viene mai modificata all'interno del corpo del `while` e la condizione del `while` è inizialmente vera, si è sicuramente in presenza di un ciclo infinito.

Un altro semplice controllo da effettuare è di verificare che l'espressione booleana che funge da condizione d'uscita non sia sempre banalmente vera, indipendentemente dal valore delle variabili che eventualmente appaiono in essa. Ad esempio, il ciclo

```
int i = 1;
while (i > 0 || i <= 10)
  i = i + 1;
```

è chiaramente senza fine (probabilmente il programmatore, inesperto, voleva dire che `i` può variare tra `1` e `10` ma ha sbagliato connettivo logico ...).

Non sempre è così immediato individuare la presenza di un ciclo senza fine. I tre frammenti di programmi C++ che seguono sono altrettanti esempi di situazioni di ciclo senza fine. Probabilmente sono tutti frutto di errori di programmazione che, purtroppo, non è così infrequente commettere ....[1] Lasciamo al lettore attento scoprire perchè si ha un ciclo senza fine e qual'è la probabile versione corretta del codice.

```
int i = 1, j = 1;
while (i = 1){
  j = j + 1;
  if (j > 10) i = 0;
}
```

[1] Si noti che si tratta di errori semantici non sintattici e quindi non rilevabili dal compilatore; in pratica, il programma non fa quello che si vorrebbe, ma è sintatticamente corretto.

(SUGG.: si ricordi che l'assegnamento può essere usato anche come espressione, che il valore `1` è interpretato come valore "vero" in un'espressione booleana, che la relazione di uguaglianza in C++ si scrive `==` e non `=`, ...).

```
int i = 1;
while (i <= 10);
  {cout << i * i << endl;
   i = i + 1;
  }
cout << "terminato!" << endl;
```

(SUGG.: si consideri che il C++ ammette anche lo *statement vuoto*, e che lo statement del `while` può essere uno statement qualsiasi (anche vuoto) ...).

```
int i = 1;
while (i <= 10)
   cout << i * i << endl;
   i = i + 1;
cout << "terminato!" << endl;
```

(SUGG.: si ricordi che il corpo del `while` è costituito da un solo statement (che però può essere anche lo statement composto) ...).

Approfondimento (Il problema della terminazione). Il problema della terminazione dei programmi è senz'altro molto complesso: a partire dagli anni '70 hanno iniziato a nascere *metodi formali* per la verifica (statica) di correttezza dei programmi che comprendono tecniche per la dimostrazione formale di proprietà di terminazione dei cicli. Per un approfondimento si veda ad esempio il testo [7].

Gli statement visti finora, in particolare gli statement `if` e `while`, sono sufficienti a scrivere qualsiasi programma (ovvero a descrivere la soluzione di qualsiasi problema computabile). In altri termini, non è strettamente necessario introdurre altri costrutti; in particolare, non è richiesta la presenza nel linguaggio di programmazione di istruzioni di salto esplicito, quali il `goto`.

Normalmente, però, i linguaggi di programmazione convenzionali offrono anche altri statement per il controllo di sequenza, quali lo statement `for`, lo `switch` (o `case`), ecc. Questi ulteriori costrutti sono introdotti, in generale, non per aumentare le capacità computazionali del linguaggio (che, come si diceva, sono già complete con i pochi tipi di statement visti fin qui), ma per motivi essenzialmente metodologici, quali:

- facilitare la scrittura dei programmi, offrendo all'utente statement che modellano in modo più naturale le strutture di controllo che si vogliono realizzare;
- migliorare la comprensibilità del programma, permettendo l'utilizzo di diverse forme di astrazione sul controllo
- aumentare l'affidabilità dei programmi, riducendo le possibilità di commettere errori grazie all'utilizzo di costrutti più specializzati, ad esempio per realizzare cicli limitati, test multipli, ecc.

Approfondimento (Teorema di Böhm-Jacopini). Il Teorema di Böhm-Jacopini, pubblicato nel 1966, afferma che qualsiasi algoritmo (o meglio qualsiasi funzione computabile) puó essere realizzata con l'utilizzo di sole tre strutture di controllo, la sequenza, la selezione e il ciclo. In termini di linguaggi di programmazione convenzionali, come il C++, questo significa che sono sufficienti i soli costrutti `if-else` e `while` (oltre alla ovvia possibilità di sequenzializzare due o più istruzioni). Questo risultato è servito, tra l'altro, a chiudere la disputa sulla necessità o meno di istruzioni di salto esplicito (come l'istruzione `goto`). Dunque qualsiasi programma può essere scritto senza l'uso di `goto`, ma usando soltanto `if-else` e `while` che hanno il pregio rispetto al `goto` di garantire la costruzione di *programmi strutturati* (vedere sottocapitolo 3.8).

3.4 Statement `do-while`

Spesso si ha a che fare con strutture di controllo iterative in cui il test d'uscita è posto alla fine del corpo del ciclo. Per modellare nel modo più naturale possibile queste situazioni, molti linguaggi di programmazione, tra cui il C++, offrono uno statement apposito. Nel caso del C++ si tratta dello statement `do-while`.

Sintassi

```
do
  S;
while (E);
```

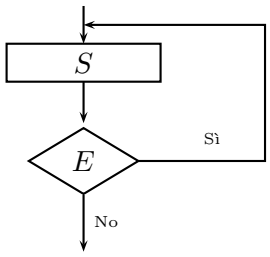
dove

E: espressione booleana;
S: statement qualsiasi.

Semantica (informale)

Esegue lo statement S; quindi valuta l'espressione E: se E ha valore "vero" allora ripete dall'inizio; altrimenti termina l'esecuzione dello statement.

La situazione è descrivibile con un diagramma di flusso nel seguente modo:



In altri termini, si esegue ripetutamente lo statement S per tutto il tempo che la condizione E rimane "vera" (si esce quando E assume valore "falso").

Si osservi che, a differenza dello statement `while`, nel `do-while` prima si esegue S e poi si valuta E. Dunque, con il `do-while`, si esegue sempre almeno un'iterazione.

Come già sottolineato, il `do-while` non è strettamente necessario e può essere sempre sostituito da un `while`. Precisamente, il generico costrutto `do-while` mostrato sopra è equivalente al frammento di programma:

S;
`while (`E`)`
  S;

Esercizio 3.11 *Realizzare il comportamento del costrutto* `while (`E`)` S`;` *tramite* `do-while`*. (SUGG.: inserire uno statement* `if` *all'inizio del corpo del* `do-while` *...).*

Due tipiche situazioni in cui l'utilizzo dello statement `do-while` risulta particolarmente comodo sono le seguenti:

Controllo dell'input: ripeti la lettura di un dato di input finchè il dato letto non soddisfa certe condizioni prestabilite. Ad esempio, se si deve leggere un numero intero `x` da standard input e si vuol imporre che il numero sia non negativo, si può sostituire la semplice istruzione `cin >> x;` con l'istruzione `do-while` seguente:

```
do
   cin >> x;
while (x < 0);
```

che impone al programma di ripetere la lettura se il dato letto è negativo (scartando di fatto tutti gli eventuali dati negativi letti). Dunque, all'uscita dal ciclo il dato contenuto in `x` sarà sicuramente non negativo.

Ripetizione programma: ripetere l'esecuzione di un programma fornendo ogni volta nuovi dati di input fintantochè l'utente non indichi esplicitamente di voler smettere.

Come esempio per illustrare questo modo di utilizzare il `do-while`, mostriamo come modificare il programma dell'Esempio 3.4 per permetterne l'esecuzione ripetuta.

Esempio 3.12 (Conversione voti → giudizi ripetuta)
Problema*. Scrivere un programma in grado di eseguire ripetutamente la conversione di un voto numerico tra 0 e 10 in un giudizio secondo lo schema indicato nell'*Esempio 3.4*. Al termine di ogni operazione di conversione, il programma dovrà richiedere all'utente se vuole continuare o no ed in caso di risposta positiva (carattere* `'s'`*) dovrà ripetere dall'inizio.*

Programma:

```
#include <iostream>
using namespace std;
int main() {
  float x;
  char ripeti;
  do {
     cout << "Dammi il voto numerico (tra 0 e 10)" << endl;
     cin >> voto;
     if (voto < 0 || voto > 10)
       cout << "voto non valido" << endl;
     else if (voto <= 5)
       cout << "insufficiente" << endl;
     else if (voto <= 6.5)
       cout << "sufficiente" << endl;
     else if (voto <= 7.5)
       cout << "buono" << endl;
     else
       cout << "ottimo" << endl;

     cout << "Altra conversione? ('s' per continuare): ";
     cin >> ripeti;
  }
  while (ripeti == 's');
  cout << "Arrivederci" << endl;
}
```

Esercizio 3.13 *Riscrivere il programma dell'Esempio 3.8 usando il costrutto* `do-while` *invece che* `while`*.*

3.5 Statement `for`

Un'altra forma molto comune di struttura di controllo iterativa è quella del cosiddetto *ciclo limitato* (o *controllato*, o *iterazione determinata*) che può essere schematicamente espressa nel modo seguente: "ripeti una certa azione per tutti i valori di x appartenenti ad un dato insieme finito di valori". Ad esempio: "stampa tutti i quadrati dei numeri interi da 1 a 100" (ovvero, "per tutti gli x tra 1 e 100 calcola e stampa $x * x$")

Il C++, come la maggior parte dei linguaggi di programmazione ad alto livello, offre uno specifico costrutto, lo statement `for`, che permette di realizzare (tra l'altro) questo tipo di cicli.

Sintassi

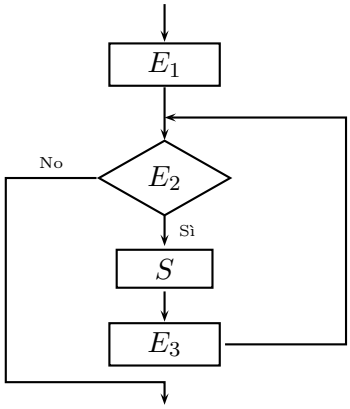
`for` (E_1; E_2; E_3)
  S;

dove

E_1, E_2, E_3: espressioni qualsiasi;
S: statement qualsiasi.

Semantica (informale)

(i) Valuta l'espressione E_1; (ii) valuta l'espressione E_2: se E_2 ha valore "falso" termina l'esecuzione del `for`; altrimenti, se E_2 ha valore "vero", esegue lo statement S e quindi valuta l'espressione E_3 e ripete da (ii).
La situazione è descrivibile con un diagramma di flusso nel seguente modo:



Nota. Il ciclo realizzato tramite `for` può sempre essere realizzato, in alternativa, utilizzando uno statement `while`. Precisamente, il generico costrutto `for` mostrato sopra è equivalente al seguente frammento di programma:

```
E1;
while (E2) {
  S;
  E3;
}
```

da cui risulta evidente, tra l'altro, che, come nel caso del `while`, potrebbe non esserci alcuna esecuzione dello statement S. Si osservi anche che le espressioni E_1 ed E_3 sono di fatto utilizzate come statement e quindi saremo verosimilmente interessati al possibile side-effect da esse provocato piuttosto che al valore ritornato dalla loro valutazione.

Questa forma molto generale di `for` può essere opportunamente specializzata per ottenere diverse forme di ciclo limitato o di struttura iterativa in genere.

Analizziamo dapprima in dettaglio come realizzare la forma base del ciclo limitato sopra menzionata; poi discuteremo brevemente le altre forme ed usi del `for`.

3.5.1 Ciclo limitato: caso base

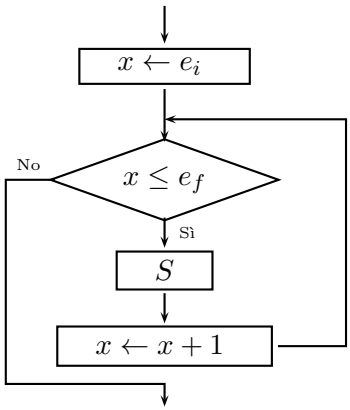
Vogliamo eseguire lo statement S per tutti i valori di una variabile x compresi tra un valore iniziale e_i ed un valore finale e_f. Per ottenere questo comportamento è sufficiente specializzare lo statement `for` nel modo seguente:

```
for (x = e_i; x <= e_f; x++)
  S;
```

dove

x:	variabile di tipo compatibile con `int`;
e_i, e_f:	espressioni di tipo compatibile con `int`;
S:	statement qualsiasi.

il cui significato è esprimibile con il seguente diagramma di flusso (in pratica si tratta di sostituire in modo opportuno le espressioni generali E_1, E_2 ed E_3 con le espressioni più specifiche che appaiono in questo uso particolare del `for`):



x costituisce la *variabile di controllo* del ciclo, mentre e_i ed e_f rappresentano, rispettivamente, il valore iniziale e finale di x. La prima espressione del `for` permette di inizializzare la variabile di controllo; la seconda costituisce la *condizione d'uscita* dal ciclo; la terza specifica il *passo* usato per l'incremento della variabile di controllo.

Il seguente frammento di programma C++ mostra un semplice esempio di utilizzo del `for` per la realizzazione di un ciclo limitato.

Esempio 3.14 *Stampa tutti i numeri interi compresi tra* 0 *e* 9 *in ordine crescente:*

```
int i;
for (i = 0; i <= 9; i++)
    cout << i << ' ';
```

Il tipo della variabile di controllo è normalmente `int`, ma può essere anche un qualsiasi tipo compatibile con `int` (ovvero un tipo *scalare*), come `char`, `bool`, un tipo definito per enumerazione, o un sottotipo di `int`, come `unsigned int`, `short`, ecc. In realtà il compilatore C++ non esegue alcun controllo sul tipo della variabile di controllo (nè tantomeno su quello delle espressioni e_i ed e_f) dato che in generale le espressioni del `for` possono essere di tipo qualsiasi. È perciò compito del programmatore fare in modo che tutte le espressioni coinvolte in un `for` usato come ciclo limitato abbiano tipo `int` o compatibile con `int`; il non rispetto di questo vincolo (utilizzando ad esempio tipi `float`) potrebbe portare alla non terminazione del ciclo.

In C++ la variabile di controllo del `for` può essere dichiarata direttamente all'interno del `for` stesso.[2] Ad esempio, il frammento di programma mostrato sopra può essere scritto alternativamente nel modo seguente:

```
for (int i = 0; i <= 9; i++) cout << i << ' ';
```

In questo caso, la variabile di controllo è *locale* al corpo dello statement `for` e quindi non può essere utilizzata al suo esterno.

Nota. Si osservi che le espressioni x `=` e_i e x`++` usate nello statement `for` sono in realtà statement di assegnamento che, come detto in precedenza, il C++ permette di usare anche come espressioni. Si osservi anche l'uso—non strettamente necessario, ma assai frequente—della forma contratta dell'assegnamento (x`++`) per realizzare l'incremento della variabile di controllo. In questo caso si potrebbe utilizzare in modo del tutto equivalente anche la forma prefissa (`++`x) in quanto qui si è interessati alla modifica della variabile x piuttosto che al valore ritornato dall'assegnamento. Dunque usare (x`++`) o (`++`x) in questo contesto è solamente una questione di stile e noi abbiamo scelto di utilizzare in questo testo sempre la forma postfissa.

Anche nel caso del `for` qualora le istruzioni all'interno del ciclo siano più di una è necessario racchiuderle in un blocco, ossia tra parentesi graffe. Vedremo altri esempi più complessi di utilizzo dello statement `for` per realizzare cicli limitati quando introdurremo le strutture dati di tipo array (si veda il sottocapitolo 4.3).

Approfondimento (Controllo ciclo limitato). Il C++ non prevede alcun controllo per garantire che un ciclo realizzato tramite `for` sia davvero un ciclo limitato e quindi termini in un numero finito di passi. In particolare in C++ è possibile modificare liberamente la variabile di controllo e i suoi limiti iniziale e finale all'interno del corpo del ciclo. Ad esempio possiamo scrivere

```
for (int i = 0; i <= 9; i++) i--;
```

che porta chiaramente ad un ciclo infinito.

Per garantire che un ciclo limitato termini sempre in un tempo finito bisognerebbe porre delle forti restrizioni sui valori iniziale e finale della variabile di controllo, sulla condizione di uscita dal ciclo e sul modo in cui la variabile di controllo viene modificata

[2] A rigore questo significa che la sintassi del `for` non è esattamente quella mostrata all'inizio di questo sottocapitolo, ma andrebbe estesa in modo da prevedere che l'espressione E_1 possa essere sostituita anche da una dichiarazione di variabile.

all'interno del ciclo stesso. In certi linguaggi di programmazione, come ad esempio il Pascal, queste restrizioni sono imposte nella forma sintattica del `for`, permettendo così di effettuare i controlli che garantiscono la finitezza del ciclo `for` già a tempo di compilazione. La scelta del C++, invece, è come al solito quella di favorire la flessibilità d'uso dei suoi costrutti, lasciando completamente al programmatore la responsabilità di garantire il funzionamento corretto dei programmi realizzati. Come detto sopra, lo statement `for` del C++ è un costrutto molto più generale del `for` di altri linguaggi, utilizzabile per realizzare svariate strutture di controllo, non soltanto il ciclo limitato.

Esercizio 3.15 *Scrivere un programma C++ che calcoli e stampi il prodotto dei primi 10 numeri naturali positivi (ovvero* 10!*) utilizzando uno statement* `for` *(SUGG.: si utilizzi una variabile* `i` *che assume, uno alla volta, i valori da 1 a 10, e per ogni nuovo valore di* `i` *si calcoli il prodotto tra* `i` *e il prodotto parziale calcolato in precedenza ed "accumulato" in una variabile* `p`*, ...).*

Esercizio 3.16 *Come per l'Esercizio 3.15, ma con il numero n dei naturali positivi da moltiplicare tra loro letto da standard input (ovvero, calcola n!). (SUGG.: il programma inizia leggendo in una variabile* `N` *il numero n, possibilmente verificando anche che sia ammissibile, e cioè* > 0*); quindi procede come nell'Esercizio 3.15, ma con il valore finale del ciclo* `for` *costituito da* `N` *(invece che da* `10`*) ...).*

3.5.2 Altri utilizzi dello statement `for`

Come più volte sottolineato, il C++ è molto permissivo riguardo all'utilizzo delle espressioni all'interno del costrutto `for`, dando così la possibilità di utilizzare questo costrutto anche per realizzare strutture di controllo diverse da quella del ciclo limitato di base visto finora.

In particolare, possiamo facilmente realizzare altre forme di ciclo limitato, ma con passo diverso da $+1$. Per ottenere questo è sufficiente specificare in modo opportuno la terza espressione (E_3) dello statement `for`. I seguenti esempi mostrano due cicli limitati, rispettivamente con passo -1 e con passo $+2$.

Esempio 3.17 *Stampa tutti i numeri interi compresi tra* 0 *e* 9 *in ordine decrescente:*

```
for (i = 9; i >= 0; i--)
  cout << i << ' ';
```

Esempio 3.18 *Stampa tutti i numeri pari compresi tra* -10 *e* 10*, in ordine crescente:*

```
for (int i = -10; i <= 10; i = i + 2)
  cout << i << ' ';
```

Le espressioni dello statement `for` possono essere anche espressioni più complesse di quelle viste finora. In particolare l'espressione E_2, che funge da condizione d'uscita dal ciclo, può essere un'espressione booleana composta costruita con i soliti connettivi logici (ad esempio, `i < 10 && i != j`). Si può anche utilizzare, al posto di una qualsiasi delle tre espressioni del `for`, la concatenazione di due espressioni ottenuta tramite l'operatore '`,`', come mostrato nel seguente esempio:

Esempio 3.19 *Stampa contemporaneamente tutti i numeri interi compresi tra* 0 *e* 9 *in ordine crescente e tra* 1 *e* 10 *in ordine decrescente:*

```
int x;
int y;
for (x = 0, y = 10; x < 10; ++x, --y)
  cout << x << ' ' << y << '\n';
```

Si osservi che è la variabile `x` *a fungere da variabile di controllo all'interno del ciclo.*

Nota. L'operatore (infisso) '`,`' permette di concatenare due o piu espressioni qualsiasi. Il significato di E_1 `,` E_2, con E_1, E_2 espressioni qualsiasi, è: valuta E_1, valuta E_2 e restituisci come risultato dell'espressione composta il valore di E_2. Ad esempio, `x = 1, y = 2, x + y` valuta le tre sottoespressioni, nell'ordine da sinistra verso destra (l'operatore '`,`' è associativo a sinistra) e restituisce come risultato finale dell'intera espressione il valore `3`.

Infine, tutte e tre le espressioni del `for` possono essere l'*espressione vuota* e quindi, di fatto, essere omesse (si noti che i "`;`" che separano le espressioni del `for` vanno comunque inseriti, anche se le espressioni sono mancanti). Ad esempio, il programma mostrato nell'Esempio 3.14 potrebbe essere scritto alternativamente come:

```
int i = 0;
for ( ; i < 10; ) {
  cout << i << ' ';
  i++;
}
```

In questo caso si sta di fatto utilizzando lo statement `for` come un `while`.

Come altro esempio, il programma mostrato nell'Esempio 3.17 può essere scritto equivalentemente nel seguente modo:

```
for (i = 10; i-- > 0; )
  cout << i << ' ';
```

In questo caso si è omessa la terza espressione del `for` ed inserito il decremento della variabile di controllo all'interno della seconda.

Per ultimo, il seguente frammento di programma realizza il calcolo del fattoriale di un numero intero `n`, utilizzando un `for` combinato con un'espressione condizionale:

```
int fatt, n;
cin >> n;
for (fatt = (n ? n : 1) ; n > 1; )
  fatt *= (--n);
```

Si tratta evidentemente di un codice particolarmente sintetico, in cui vengono sfruttate varie peculiarietà del C++ (che lasciamo da scoprire ed analizzare al lettore attento). È altrettanto evidente, comunque, che la sinteticità è ottenuta a discapito della chiarezza del codice.

La possibilità di omettere le espressioni del `for` rende ancor più evidente il fatto che con il `for` del C++ è possibile scrivere cicli infiniti. In particolare, se manca l'espressione E_2 del `for` e non sono presenti interruzioni forzate del ciclo (per esempio attraverso l'utilizzo di un `break`), allora il `for` provoca sicuramente un ciclo infinito.

L'utilizzo di queste forme particolari di `for` può rendere il programma più difficile da comprendere ed è quindi, in generale, sconsigliato. Come regola generale, è sempre preferibile utilizzare il costrutto sintattico che modella in modo più naturale la struttura di controllo (iterativa) che si deve realizzare, sebbene la flessibilità del linguaggio di programmazione permetta di utilizzare indifferentemente uno qualsiasi dei costrutti offerti.

3.6 Statement `switch`

Come abbiamo già avuto modo di evidenziare, una struttura di controllo molto comune è quella del test multiplo. In particolare, è frequente la situazione in cui si deve scegliere di eseguire una fra k possibili azioni in base ad m valori distinti ($m \geq k$) che una data espressione E può assumere. Questa situazione può essere modellata tramite `if-else` annidati. Ma molti linguaggi, tra cui il C++, offrono un apposito costrutto sintattico (`case`, `switch`, ...) che permette di modellare in modo più naturale ed immediato questa forma di controllo del flusso.

Nel caso del C++ lo statement in questione è lo `switch`. Vediamo come utilizzare questo statement per realizzare la struttura di controllo di test multiplo sopra descritta, nel caso $m = k$.

Sintassi

```
switch (E) {
case C_1:
   S_Seq_1; break;
case C_2:
   S_Seq_2; break;
...
case C_k:
   S_Seq_k; break;
default:
   S_Seq;
}
```

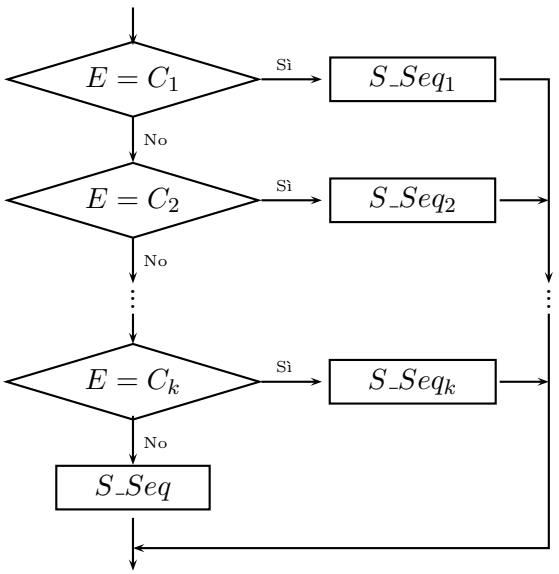
dove

E:	espressione di tipo compatibile con `int`;
C_1, C_2, ..., C_k ($k \geq 0$):	espressioni costanti di tipo compatibile con `int`;
S_Seq_1,..., S_Seq_k, S_Seq:	sequenze di statement qualsiasi.

Semantica (informale)

Valuta l'espressione E; se esiste i, con $i = 1, \ldots, k$, tale che il valore di E è uguale al valore di C_i, allora esegue la sequenza di statement S_Seq_i e quindi termina; altrimenti, esegue lo statement S_Seq, se presente, e quindi termina.

La situazione è descrivibile con un diagramma di flusso nel seguente modo:



Si tratta di un test multiplo in cui si sceglie una tra k possibili sequenze di statement S_Seq_1, ..., S_Seq_k, in base al valore di un'espressione E. La parte introdotta dalla parola chiave `default` è opzionale e viene eseguita soltanto se il valore di E non combacia con nessuna delle costanti specificate nei rami `case`.[3]

Il seguente frammento di programma mostra un semplice esempio di questo particolare utilizzo dello statement `switch`. Scopo del programma è di stampare il nome italiano dell'operatore aritmetico rappresentato dal carattere contenuto nella variabile `c`.

```
...
char c;
switch (c) {
case '+':
   cout << c << " -> addizione" << endl; break;
case '-':
   cout << c << " -> sottrazione" << endl; break;
case '*':
   cout << c << " -> moltiplicazione" << endl; break;
case '/':
   cout << c << " -> divisione" << endl; break;
default:
   cout << "Non e' un operatore aritmetico!" << endl;
}
```

In questo esempio (come spesso accade), l'espressione di controllo è semplicemente una variabile singola.

Si osservi che le costanti $C_1, \ldots, C_k$ non possono essere espressioni qualsiasi, ma devono essere necessariamente espressioni *costanti*, e cioè espressioni il cui valore è determinabile a tempo di compilazione (ad esempio `2` o `2+3`, ma non `2+x`, con `x` variabile). Inoltre i valori di $C_1, \ldots, C_k$ devono essere tutti diversi tra di loro (in caso contrario il compilatore segnala un errore). Infine, il tipo di $C_1, \ldots, C_k$ e quello di E devono essere compatibili tra loro e tutti di tipo scalare (e cioè, compatibile con `int`).

Per quanto riguarda il "corpo" di ogni alternativa `case` si osservi che si tratta di una sequenza di statement e non di un singolo statement: questo implica tra l'altro che, nel caso il corpo dell'alternativa debba contenere più statement, non è necessario racchiuderli tra parentesi graffe in modo da ottenere lo statement composto.

Esercizio 3.20 *Scrivere un programma C++ che legge da standard input un numero intero e stampa il corrispondente mese dell'anno o un opportuno messaggio di errore se il numero non è compreso tra 1 e 12.*

[3] Se presente, l'alternativa `default` deve essere unica, e può comparire ovunque nello `switch` (anche se è prassi inserirla sempre come ultima alternativa).

Approfondimento (Implementazione dello `switch`). Lo statement `switch` viene normalmente implementato in modo particolarmente efficiente su una macchina convenzionale. Senza entrare nei dettagli di un'implementazione reale, si può pensare che le costanti delle diverse alternative `case` vengano viste a livello di linguaggio Assembly come altrettante *etichette* associate al codice della sequenza di statement che segue il relativo `case`. L'esecuzione di uno `switch` allora procede valutando dapprima l'espressione di controllo dello `switch` stesso e quindi eseguendo un salto al codice che ha come etichetta il valore dell'espressione appena calcolato. Questa tecnica di implementazione spiega varie caratteristiche dello `switch` tra cui il fatto che le costanti associate ai `case` debbano essere valutabili a tempo di compilazione e debbano essere tutte diverse tra loro. Inoltre si osservi che il codice relativo alle sequenze di statement dei diversi rami `case` sarà disposto in memoria in modo consecutivo, in base all'ordine che gli statement hanno nello `switch`, e che quindi, una volta effettuato un salto alla porzione di codice individuata dall'etichetta corrispondente al valore assunto dall'espressione di controllo, l'esecuzione prosegue sul codice che segue, a meno che non sia prevista un'istruzione di salto esplicito che permette di trasferire il controllo oltre il codice dello `switch`.

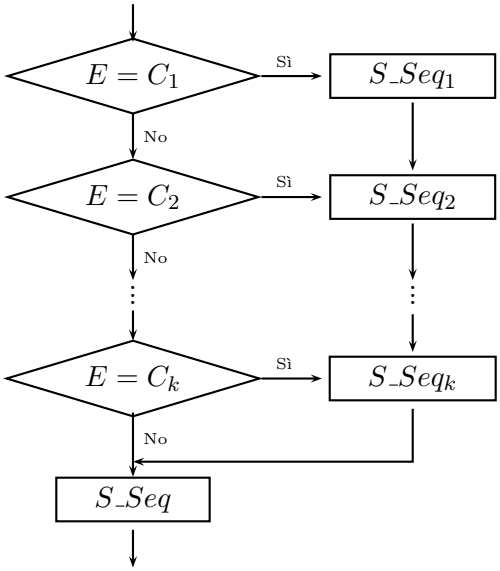
Nella forma di `switch` fin qui considerata, ogni alternativa, tranne l'ultima, termina con uno statement `break`. L'esecuzione di questo statement—su cui ritorneremo in modo più approfondito nel sottocapitolo successivo—provoca la terminazione immediata dello statement `switch` (e quindi la prosecuzione dell'esecuzione dallo statement successivo lo `switch`).[4]

In realtà lo statement `switch` del C++ non richiede, in generale, che un'alternativa `case` termini necessariamente con uno statement `break`: la sequenza di statement successivi alla parola chiave `case` può essere una sequenza qualsiasi (anche vuota).

Dunque, nel caso più generale (cioè senza i `break`), la semantica dello `switch` diventa: valuta l'espressione E; se esiste i, con $i = 1, \ldots, k$, tale che il valore di E è uguale al valore di C_i, allora esegue nell'ordine le sequenze di statement S_Seq_i, $S\_Seq_{i+1} \ldots, S\_Seq_k$ e quindi termina; altrimenti, esegue lo statement S_Seq, se presente, e quindi termina.

Con un diagramma di flusso la situazione è rappresentata nel modo seguente:

[4] Si noti che dopo uno statement `break` in un'alternativa `case` non ha senso che compaiano altri statement all'interno della stessa alternativa in quanto non potrebbero mai essere eseguiti.



In altri termini, l'esecuzione del corpo dello `switch` comincia, in generale, dall'alternativa `case` a cui è associato un valore uguale al valore della condizione dello `switch` e continua attraverso le successive alternative fino a che non termina lo `switch` o incontra un `break` (o un'altra istruzione che provoca forzatamente il salto fuori dallo `switch`). Si tratta dunque di una struttura di controllo decisamente diversa da quella del test multiplo descritto all'inizio del paragrafo sottocapitolo.

Riferendoci al frammento di programma mostrato sopra, l'assenza dell'istruzione `break` all'interno per esempio del `case '*'` provocherebbe come output

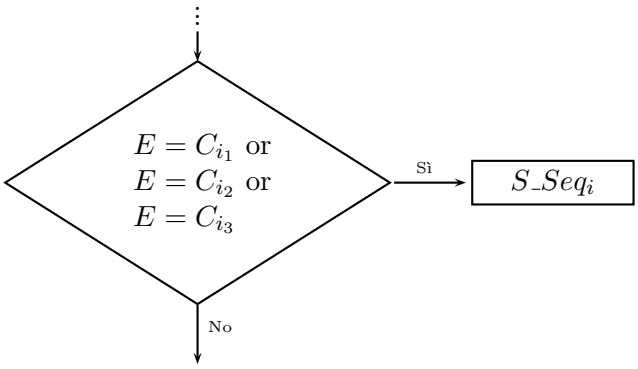
```
* -> moltiplicazione
* -> divisione
Non e' un operatore aritmetico!
```

in risposta al carattere `'*'` fornito come input che, evidentemente, non è quanto si voleva ottenere. Si noti come la mancanza dei `break` renda significativo l'ordine in cui appaiono le diverse alternative dello `switch`.

Nota. Il costrutto `switch` viene utilizzato nella maggior parte dei casi per realizzare la struttura di controllo di test multiplo considerata all'inizio del sottocapitolo, che richiede necessariamente l'utilizzo di un `break` alla fine di ogni ramo `case` dello `switch` (tranne l'ultimo che non necessita del `break` non essendo seguito da altre alternative `case`). Dunque l'assenza di un `break` è molto probabilmente segnale di un possibile errore e quindi si consiglia di evidenziare con un commento un'eventuale omissione voluta di un `break`.

La forma più generale dello `switch` permette facilmente di modellare anche la situazione di test multiplo in cui ci siano più valori possibili in corrispondenza a ciascuna alternativa dello `switch` (caso $m > k$). Ad esempio

supponiamo che la sequenza di statement S_Seq_i, $1 \leq i \leq k$, debba essere eseguita quando l'espressione E assume uno tra tre possibili valori, C_{i_1}, C_{i_2}, C_{i_3}, e cioè, con un diagramma di flusso:



Per modellare questa situazione basta inserire nello `switch` tre alternative `case` consecutive nel modo seguente:

```
switch (E) {
...
case C_{i_1}:
case C_{i_2}:
case C_{i_3}:
   S_Seq_i; break;
...
}
```

Si noti che i primi due `case` sono alternative dello `switch` con la parte relativa alla sequenza di statement vuota (in particolare, senza `break`) e quindi la loro esecuzione comporta semplicemente la prosecuzione sullo statement dell'alternativa successiva.

Esempio 3.21 (Conta numero vocali)
Problema. *Scrivere un programma che legge da standard input una sequenza di caratteri terminata da un punto, determina il numero di vocali (maiuscole o minuscole) presenti nella sequenza e quindi stampa il numero di vocali presenti per ciascuna delle 5 vocali (si veda per confronto il programma dell'Esempio 3.8).*

Programma:

```
#include <iostream>
using namespace std;
int main() {
  char c;
  int n_a = 0, n_e = 0, n_i = 0, n_o = 0, n_u = 0;
```

```
  cout << "Inserisci sequenza di caratteri terminata da ."
       << endl;
  cin >> c;   //legge un carattere e lo assegna a c
  while (c != '.'){
    switch(c)  {
    case 'a': case 'A':
      n_a++; break;
    case 'e': case 'E':
      n_e++; break;
    case 'i': case 'I':
      n_i++; break;
    case 'o': case 'O':
      n_o++; break;
    case 'u': case 'U':
      n_u++;
    }
    cin >> c;
  }
  cout << "La sequenza data contiene " << endl;
  cout << n_a << " vocali a" << endl;
  cout << n_e << " vocali e" << endl;
  cout << n_i << " vocali i" << endl;
  cout << n_o << " vocali o" << endl;
  cout << n_u << " vocali u" << endl;
  return 0;
}
```

Approfondimento (`switch` vs `if`). Uno statement `switch` può essere facilmente sostituito da una serie di statement `if-else` annidati. Ad esempio, la situazione descritta dallo statement `switch`

```
switch (E) {
case C1: case C2: case C3:
   S_Seq1; break;
case C4:
   S_Seq2; break;
default:
   S_Seq;
}
```

può essere realizzata in modo equivalente (almeno per quanto riguarda il comportamento esterno) tramite `if-else` annidati nel modo seguente:

```
if (E == C1 || E == C2 || E == C3) {S_Seq1}
else if (E == C4) {S_Seq2}
else {S_Seq}
```

Il codice scritto utilizzando il costrutto `switch` presenta alcuni vantaggi rispetto al codice che utilizza `if-else` annidati:

- risulta, in generale, più leggibile;
- può essere eseguito, in generale, in modo più efficiente grazie alla tecnica di implementazione dello `switch` che richiede sempre un solo test per individuare l'alternativa da selezionare (per contro, con gli if-else annidati, se l'alternativa è l'n-esima verranno effettuati n test prima di poterla selezionare).

Si osservi che il test usato per selezionare un'alternativa in uno statement `switch` è necessariamente un test di uguaglianza (il valore di E è uguale al valore di C_i?). Questo rende molto più problematico realizzare tramite `switch` un test multiplo in cui le condizioni di selezione delle diverse alternative siano in generale delle disuguaglianze. Ad esempio, una situazione del tipo *"se* $C_1 \leq E < C_2$ *allora esegui* S_1*; se* $C_2 \leq E < C_3$ *allora esegui* S_2*; altrimenti esegui* S_3*"* viene realizzata in modo molto più naturale tramite `if-else` annidati (si veda ad esempio il programma di conversione di voti in giudizi mostrato nell'Esempio 3.4).

Nota. Si osservi che è possibile ottenere il comportamento di un generico `if-else`

```
if (E) S1
else S2
```

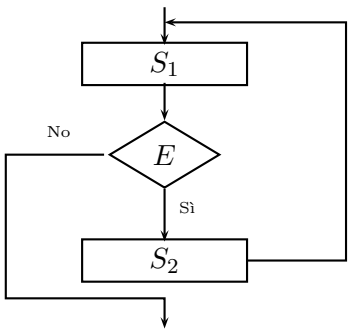
utilizzando uno statement `switch` nel modo seguente:

```
switch (E) {
   case true: S1; break;
   case false: S2: break;
}
```

È chiaro che si tratta di un "trucco" di programmazione che sicuramente non contribuisce alla chiarezza del programma e che quindi andrebbe di norma evitato.

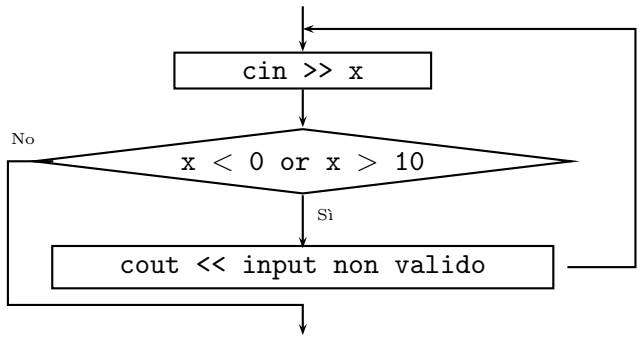
3.7 Statement `break`

Finora abbiamo visto strutture di controllo per l'iterazione non limitata in cui la condizione d'uscita dal ciclo è posta o all'inizio o alla fine del ciclo, e che possono essere realizzate in modo naturale rispettivamente tramite gli statement `while` e `do-while`. Nella pratica, però, sono frequenti anche casi in cui la condizione d'uscita dal ciclo è posta in mezzo al ciclo, cioè preceduta e seguita da altre azioni, come illustrato nel seguente diagramma di flusso:



Una simile struttura di controllo può ovviamente essere realizzata con gli statement `while` e `do-while`, ma, come vedremo, in modo non del tutto

naturale. Si consideri ad esempio la sequenza di istruzioni descritta dal seguente diagramma di flusso:



che rappresenta una tipica situazione di lettura di un dato di input con controllo di validità del dato stesso e ripetizione della lettura in caso di dato non valido (preceduta da un opportuno messaggio d'errore). Questo schema può essere realizzato ad esempio utilizzando un costrutto `do-while` nel modo seguente:

```
int x;
do {
  cin >> x;
  if (x < 0 || x > 10)
    cout << "input non valido";
}
while (x < 0 || x > 10);
```

È evidente che questo codice realizza una struttura di controllo diversa (e senz'altro più complicata) rispetto a quella indicata sopra anche se il comportamento esterno delle due è lo stesso (lasciamo al lettore come esercizio la descrizione di questa struttura di controllo tramite un diagramma di flusso).

In casi come questo può essere comodo avere a disposizione uno statement che permetta di "saltar fuori" da un ciclo in un punto qualsiasi del ciclo stesso. Il C++ offre una simile possibilità attraverso lo statement `break`.

La semantica dell'istruzione `break` è quella del salto: l'esecuzione di un `break` provoca un salto al primo statement successivo al costrutto che lo contiene. Come costrutti contenenti il `break` si considerano soltanto quelli iterativi `while`, `do-while`, `for` e lo `switch` (ma non lo statement `if` o lo statement composto). L'utilizzo di un `break` all'interno di questi costrutti causa l'immediata uscita dal costrutto stesso, mentre il suo utilizzo all'esterno di questi costrutti (ad esempio all'interno di uno statement `if` non contenuto a sua volta all'interno di un ciclo o di uno `switch`) viene segnalato come errore dal compilatore.

Utilizzando il `break` è possibile riscrivere il codice dell'esempio mostrato sopra in modo che rispecchi in modo più preciso la struttura di controllo desiderata:

```
int x;
do {
  cin >> x;
  if (x < 0 || x > 10) cout << "input non valido";
  else break;
}
while (true);
```

Si osservi che il `break` comporta l'uscita dal costrutto `do-while`, ignorando il fatto che il `break` appare all'interno di un `if-else`. Si osservi anche che nel caso in cui il `break` sia all'interno di più costrutti iterativi o `switch` annidati la sua esecuzione provoca l'uscita soltanto dal costrutto che lo contiene direttamente e non da quelli più esterni.

Vediamo ora un altro esempio (un programma completo) in cui si utilizza lo statement `break`.

Esempio 3.22 (Conta il numero di lettere 'a')
Problema. *Leggere da standard input una sequenza di caratteri (di lunghezza massima 32) terminata da spazio o da "a capo" e determinare il numero di lettere 'a' presenti. Scrivere quindi il risultato su standard output. Nel caso in cui venga raggiunta la lunghezza massima della sequenza stampare anche un opportuno messaggio d'avviso.*

Programma:

```
#include <iostream>
using namespace std;
int main() {
  int i = 0, num_a = 0;
  cout << "Inserisci una sequenza di caratteri "
       << " terminata da 'spazio' o da 'a capo'" << endl;
  char c = cin.get();
  ++i;
  while (c != ' ' && c != '\n') {
    if (i > 32) {
      cout << "Attenzione: la sequenza e' stata troncata!"
       << endl;
      break;
    }
    if (c == 'a')
      ++num_a;
```

```
    c = cin.get();
    ++i;
  }
  cout << "Il numero di caratteri 'a' e' " << num_a << endl;
  return 0;
}
```

Si presti attenzione al fatto che un utilizzo indiscriminato del `break` può portare a programmi più difficili da capire (e da dimostrare corretti). Risulta infatti più difficile, in generale, individuare e quindi controllare la condizione di uscita da ciclo che, con l'uso del `break`, piò essere "distribuita" in più parti del costrutto che realizza il ciclo. L'uso del `break` dovrebbe pertanto essere limitato ai soli casi in cui è evidente un effettivo miglioramento nella leggibilità del programma, come mostrato, ad esempio, nelle situazioni presentate all'inizio del sottocapitolo.

Esercizio 3.23 *Riscrivere il programma dell'Esempio 3.22 senza utilizzare l'istruzione* `break`*. (SUGG.: aggiungere la condizione* `i <= 32` *all'interno dell'espressione booleana di controllo del ciclo* `while` *e portare il test per determinare se la sequenza è stata troncata dopo lo statement* `while`*).*

3.8 Statement `goto` e programmazione strutturata

Il C++, così come la maggior parte dei linguaggi di programmazione convenzionali, offre anche un'istruzione di salto "incondizionato": lo statement `goto`.

La sintassi del `goto` è la seguente:

```
goto L
```

dove `L` è un identificatore associato ad uno statement `s` nel modo seguente:

```
L: s;
```

`L` è detta l'*etichetta* (in inglese *label*) dello statement `s`.

Il significato dello statement `goto L` è il seguente: l'esecuzione del programma continua dallo statement identificato dall'etichetta `L`.

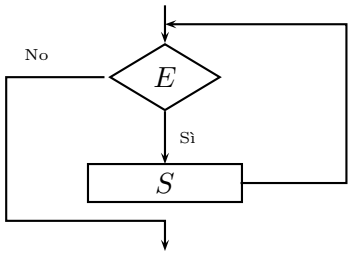
Esempio 3.24

```
Ripeti: if (x<0) {
              s=s+x;
              x=x+1;
              goto Ripeti;
        }
```

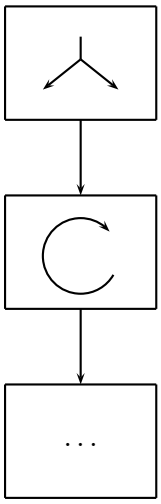
Si osservi che questa sequenza di istruzioni realizza la stessa struttura di controllo realizzata dal costrutto `while`.

Un'etichetta può essere associata a qualsiasi statement in un programma e deve essere unica. Un salto tramite `goto` può avvenire sia all'indietro che in avanti, e cioè sia ad un'etichetta che precede, nel testo del programma, lo statement `goto`, che ad un'etichetta che lo segue (nel caso di salto in avanti, si noti che l'etichetta viene utilizzata nel `goto` prima di essere stata introdotta). Inoltre con il `goto` è possibile saltare fuori/dentro cicli. Non è invece possibile saltare fuori/dentro una funzione.

I costrutti utilizzati finora, e cioè i costrutti `while`, `if-else`, `do-while`, ecc., permettono di realizzare soltanto strutture di controllo con *un unico punto di ingresso e un unico punto di uscita*. Ad esempio, con il `while` si realizza la struttura di controllo a un ingresso e una uscita descritta dal seguente diagramma di flusso:



Un programma completo è ottenuto "connettendo" l'uscita di una struttura di controllo all'entrata della successiva. Il flusso di controllo del programma risulta pertanto molto semplice, lineare:



Ciascun costrutto al suo interno potrà avere una struttura di controllo relativamente complessa (biforcazioni, cicli, ...), ma comunque non visibile all'esterno.

La situazione può risultare drasticamente diversa con l'uso dell'istruzione `goto`. In questo caso si possono realizzare strutture di controllo con più uscite e più entrate. Si consideri ad esempio il seguente frammento di codice C++ in cui si fa un uso indiscriminato di `goto`.

```
C: if (x > 0) goto A;
   else goto B;
A: if (y < 0) goto C;
   else goto B;
B: ...
```

E' evidente che in situazioni del genere il flusso di controllo del programma risulta molto più complesso, non descrivibile in generale come una semplice sequenza di costrutti connessi da un solo arco, ma piuttosto come un grafo complicato, con più archi uscenti da ogni costrutto. La situazione che si può creare è resa bene dal nome che è stato attribuito ad un codice di questo tipo: *"codice spaghetti"*.

3.8.1 Programmazione strutturata

L'uso di strutture di controllo con una sola entrata e una sola uscita (senza l'uso di `goto`) è una delle caratteristiche fondamentali della cosiddetta programmazione strutturata.

La *programmazione strutturata* è una metodologia di programmazione, sviluppata negli anni '70-'80, allo scopo di rendere la struttura degli algoritmi realizzati dai programmi più facile da capire e da modificare.

In particolare, la programmazione strutturata prevede l'uso *disciplinato* di:

- strutture di controllo, che devono essere soltanto del tipo a 1-entrata 1-uscita;
- convenzioni sul *formato del programma*, ovvero sulla forma grafica del codice sorgente (ad esempio, l'uso opportuno di "indentatura");
- commenti e altre convenzioni lessicali e grafiche (ad esempio, l'uso di nomi di variabili significativi);
- opportune astrazioni sui tipi di dato, con l'utilizzo di tipi di dato definiti da utente.

I vantaggi dell'utilizzo della programmazione strutturata si possono riassumere nei seguenti punti:[5]

- migliora la leggibilità (ovvero la comprensibilità) del programma
- semplifica l'individuazione di errori e la verifica (formale) di correttezza del programma (ad esempio, con l'utilizzo di pre- e post-condizioni, invarianti di ciclo)
- semplifica la manutenzione del programma.

E' importante notare che la programmazione strutturata—ed in particolare l'utilizzo di sole strutture di controllo a 1-entrata 1-uscita—non limita in nessun modo il potere computazionale del linguaggio. Vale infatti il risultato enunciato nel già citato Teorema di Böhm-Jacopini (vedi sottocapitolo 3.3) in base al quale le strutture di controllo `if-else` e `while`, più la sequenza, sono un insieme di strutture "completo", nel senso che sono sufficienti a scrivere qualsiasi programma (dunque, in particolare, il `goto` non è necessario).

Va osservato, comunque, che a volte l'uso di strutture di controllo a 1-entrata 1-uscita può risultare un pò forzato e di fatto complicare la struttura dell'algoritmo invece che semplificarla. Inoltre il codice strutturato potrebbe risultare meno efficiente dal punto di vista dei tempi d'esecuzione di un codice equivalente, ma "meno strutturato" (problematico per alcune applicazioni real-time).

In generale, i linguaggi di programmazione possono imporre più o meno il rispetto dei principi della programmazione strutturata, in particolare per quanto riguarda le strutture di controllo. Ad esempio il Pascal evita il più possibile l'utilizzo di costrutti linguistici che possano portare a strutture di controllo non ben strutturate. Il C++ invece è, come sempre, più permissivo e offre, oltre al `goto`, altre possibilità che facilitano la programmazione, ma possono portare a codice non ben strutturato (ad esempio, l'uso senza vincoli dello statement `return`—vedi Capitolo 5).

Nel seguito dunque eviteremo di usare il `goto` in tutti i programmi che realizzeremo, che però potrebbero risultare, per motivi di convenienze, non sempre rigidamente ben strutturati, dal punto di vista della struttura di controllo. Cercheremo invece di attenerci sempre il pirù possibile alle altre indicazioni previste nella programmazione strutturata.

[5]Si osservi che in questa discussione stiamo assumendo la fondamentale proprità che per uno stesso algoritmo si possano avere diversi programmi equivalenti, che potranno quindi essere piò o meno strutturati. Due programmi si dicono equivalenti se con gli stessi dati di input o non terminano o se terminano producono lo stesso effetto.

3.9 Controllo dei dati in input

Come indicato nel paragrafo 2.8.1, il dato letto tramite l'operatore di estrazione `s >> v`, con `s` stream di input e `v` variabile di tipo `t`, deve essere una costante di tipo `t`, sintatticamente corretta. Altrimenti, il dato viene rifiutato (alla variabile v non viene assegnato alcun valore) e lo stream `s` viene impostato allo stato `failed`. È possibile verificare la presenza di questo stato tramite la funzione di libreria `fail`, che funziona nel modo seguente:

```
s.fail()
```

restituisce `true` se lo stream `s` si trova in uno stato `failed`, `false` altrimenti.

Esempio 3.25

```
cout << "inserire un numero intero";
int n;
cin >> n;
if (cin.fail())
  cout << "errore nel dato di input" << endl;
else
  cout << "il dato inserito e' corretto" << endl;
```

Una volta individuato un errore nell'inserimento del dato, è frequente richiedere all'utente di inserire nuovamente il dato. Per far questo bisogna provvedere prima a ripristinare lo stato corretto dello stream di input `s` tramite la funzione di libreria `clear`, nel modo seguente:

```
s.clear()
```

Purtroppo questo non è ancora sufficiente. Infatti sullo stream di input è presente ancora il dato che l'operatore `>>` non è riuscito a leggere. Prima di ripetere la lettura bisogna pertanto rimuovere questo dato dallo stream, altrimenti la lettura fallisce nuovamente (andando quindi in loop). Un modo comodo per ottenere lo "svuotamento" dello stream di input è tramite la funzione di libreria `ignore`, che funziona nel modo seguente:

```
s.ignore(n,delim)
```

estrae i caratteri dallo stream di input `s` e li scarta; l'estrazione termina quando sono stati estratti e scartati `n` caratteri o quando si è trovato il carattere `delim` (dipende da quale delle due condizioni avviene per prima); nell'ultimo caso anche il carattere delimitatore viene estratto.

Esempio 3.26

```
cout << "inserire un numero intero";
int n;
do {
  cin >> n;
  if (cin.fail()) {
     cout << "errore - ripetere" << endl;
     cin.clear();
     cin.ignore(256,'\n');
  }
  else break;
}
while (true);
cout << "il dato inserito e' corretto" << endl;
```

`256` *è un numero arbitrario; in questo modo si fa l'ipotesi di poter leggere e scartare, per ogni lettura che sia fallita, al massimo 256 caratteri consecutivi.*

Modi alternativi per il test dello stream

Per verificare lo stato di uno stream è possibile usare direttamente il nome dello stream come condizione booleana.

Esempio 3.27

```
...
int n;
cin >> n;
if (cin)
   cout << "il dato inserito e' corretto" << endl;
else
   cout << "errore nel dato di input" << endl;
```

Nota Questo è possibile perchè la classe `istream` fornisce una funzione che può convertire un oggetto di tipo `istream`, come ad es. `cin`, in un valore booleano. Questa funzione viene chiamata quando lo stream (ad es. `cin`) appare in una posizione in cui è richiesta un'espressione boolena, come ad esempio nella condizione di un `if` o di un `while`.

Un altro modo alternativo per ottenere lo stesso risultato è di controllare direttamente il risultato della valutazione dell'operatore `>>`. Siccome questo operatore restituisce come suo risultato uno stream (precisamente, lo stream modificato a seguito dell'estrazione del dato di input), è possibile testare lo stream risultante esattamente come nella prima soluzione alternativa.

Esempio 3.28

```
...
int n;
if (cin >> n)
   cout << "il dato inserito e' corretto" << endl;
else
   cout << "errore nel dato di input" << endl;
```

L'uso della funzione `fail` risulta semanticamente più chiaro, ma spesso si preferiscono le soluzioni alternative perchè più sintetiche.

Nota A rigore l'uso degli stream come condizione di test risulta un pò più generale dell'uso del `fail` perchè permette di testare anche altre possibili cause di fallimento, come ad esempio errori di disco.

3.10 Regole di "scope"

Una dichiarazione (di variabile, di costante, di tipo, di funzione, ...) introduce un legame ("binding") tra un *nome* ed un *oggetto denotabile*.

I linguaggi di programmazione di solito permettono di utilizzare lo stesso nome, in contesti diversi, per denotare oggetti diversi. Un'associazione nome—oggetto, dunque, non rimane attiva necessariamente per tutto il programma, ma può essere attiva in certi punti del programma e non attiva in altri, e in quest'ultimi potrebbe essere attiva invece un'associazione relativa allo stesso nome, ma con un oggetto diverso.

Si pone quindi il problema quando si incontra un nome nel programma, di capire a quale dichiarazione (e quindi a quale oggetto denotabile) esso si riferisce.

La risposta a questa domanda è fornita dalle regole di "scope": regole che permettono di determinare il *campo d'azione* (o "scope") di ogni dichiarazione (ovvero, in quale parte del programma è *attiva* quella dichiarazione), in modo tale che ogni nome che si incontra nel programma sia associato ad una ed una sola dichiarazione.

Nei punti del programma in cui una dichiarazione per un nome N è attiva diremo che il nome N è *visibile*.

Nei linguaggi di programmazione convenzionali le regole di "scope" sono solitamente basate sulla nozione di *blocco* (regole di "scope" statiche).

Blocco: parte del programma (= insieme di dichiarazioni e statement) delimitata da opportuni costrutti sintattici (ad esempio, parentesi graffe aperte e chiuse, coppie di parole chiave `begin-end`, ecc.). □

Per esempio, lo statement composto del C++

```
{
   dich.
   +
   stmt
}
```

definisce un blocco. Dunque, il corpo del main (e più in generale, come vedremo in seguito, di una qualsiasi funzione), così come il corpo di uno statement strutturato, costituiscono altrettanti blocchi.

Si noti che nel caso dello statement `for` del C++ si considerano parte del blocco del `for` anche le tre espressioni che compaiono nella testata del costrutto `for`. Ad esempio.:

```
for(int i=0; i<n; i++)
  {...}
```

il blocco di questo statement `for` è costituito dalle dichiarazioni e statement presenti nel suo corpo + la dichiarazione ed espressioni presenti nella testata del `for`.

Nota. Non è così ad esempio per il `do-while`; ad es. nello statement:

```
do {
    ...
} while (i < 10);
```

l'espressione `i < 10` è esterna al blocco rappresentato dal corpo del `while`.

Inoltre, per comodità di trattazione, indicheremo come *blocco del programma* l'insieme delle eventuali dichiarazioni poste nella parte più esterna

del programma (e quindi non contenute nel blocco del `main` né di nessun'altra funzione).

Esempio 3.29 (Blocchi in un programma)

```
const int max = 100;
int main(...)
    { ...
      while(...)
       {... }
      ...
    }
```

In questo programma si evidenziano tre blocchi:

- *blocco del programma, che contiene una dichiarazione per la costante* ***max***
- *blocco del* ***main****, contenuto nel blocco del programma (= blocchi annidati)*
- *blocco del* ***while****, contenuto nel blocco del* ***main****.*

Regole di "scope". Il campo d'azione di una dichiarazione per un nome `N` contenuta in un blocco `B` si estende dal punto in cui essa compare fino al termine del blocco `B` stesso. Il campo d'azione della dichiarazione si estende anche agli eventuali blocchi contenuti in `B` (sotto-blocchi), a meno che quest'ultimi non contengano a loro volta una dichiarazione per lo stesso nome `N`, nel qual caso il campo d'azione della dichiarazione in `B` è sospeso fino al termine del sotto-blocco che la contiene. □

Dunque le regole di "scope" stabiliscono che un nome `N` è visibile all'interno del blocco in cui dichiarato, ma non visibile al suo esterno. Inoltre, nel caso in cui un sotto-blocco contenga una nuova dichiarazione per `N`, la nuova dichiarazione "nasconde" temporaneamente quella vecchia.

Nota. Vale comunque sempre la regola che un nome `N` deve essere in ogni istante associato ad uno ed un solo oggetto denotabile. Quindi, ad esempio, la presenza di due dichiarazioni per lo stesso nome `N` all'interno dello stesso blocco viene segnalata (dal compilatore) come un errore.

Viceversa, uno stesso oggetto può essere associato in un certo istante a più nomi ("*aliasing*"). Questo capita ad esempio con il passaggio parametri per riferimento (si veda sottocapitolo 5.5). Se una funzione `f` ha un parametro formale `x` passato per riferimento e si richiama `f` specificando come parametro attuale una variabile `a`, all'interno della funzione `f` il nome `x` sarà associato allo stesso oggetto denotato da `a`. Quest'ultima associazione verrà eliminata nel momento in cui la funzione `f` terminerà, mentre verrà mantenuta l'associazione con il nome `a`.

Esempio 3.30 (Regole di scope)

```
const int max = 100;
int n = max;
int main() {
    int x = 0;
    int y = 1
    cout << n;
    cout << x;
    for (int i=0; i<n; i++) {
       int x = 1;
       cout << x << endl;
       x = y + 1;
    }
    cout << x;
    cout << i; // errore!
    return 0;
}
```

Il campo d'azione delle dichiarazioni per `max` *e* `n` *è l'intero programma. Il campo d'azione della prima dichiarazione per* `x` *è il* `main`*, però con un "buco" dovuto alla presenza all'interno del sotto-blocco del* `for` *di un'altra dichiarazione per lo stesso nome* `x`*. In questo sotto-blocco la prima dichiarazione per* `x` *non è più attiva; risulta invece attiva la seconda dichiarazione per* `x`*, quella interna al corpo del* `for`*. Dunque la stampa di* `x` *fatta all'interno del* `for` *fa riferimento alla* `x` *"locale" al* `for` *stesso (e quindi il primo valore stampato sarà* 1*).*

Nel blocco del `for` *non c'e' alcun modo per riferirsi all'oggetto creato con la prima dichiarazione per* `x`*. Quest'ultima ritonerà attiva invece al termine del blocco del* `for` *nel momento in cui si rientra nel blocco del* `main`*. L'istruzione di stampa di* `x` *successiva al blocco del* `for` *perciò farà riferimento alla* `x` *dichiarata nel* `main` *e quindi produrrà il valore* `0`*.*

La successiva istruzione di stampa della variabile `i` *invece viene rilevata dal compilatore come errata in quanto il nome* `i` *a cui si fa riferimento in essa non è visibile in quel punto, essendo* `i` *dichiarato in un blocco più interno.*

L'insieme delle associazioni nomi–oggetti denotabili esistenti in un certo momento dell'esecuzione in un dato punto del punto del programma è detto *ambiente di riferimento* (o, semplicemente, *ambiente*). Le associazione nomi–oggetti denotabili sono introdotte nell'ambiente in particolare tramite le dichiarazioni.

Ad esempio, nel programma mostrato sopra, nel punto immediatamente precedente il `for`, l'ambiente è costituito dalle associazioni introdotte dalle dichiarazioni:

- per `max` e `n`, contenute nel blocco del programma
- per `x` e `y`, contenute nel blocco del main.

All'interno del `for`, dopo la dichiarazione di `x`, invece, l'ambiente è costituito dalle associazioni introdotte dalle dichiarazioni:

- per `max` e `n`, contenute nel blocco del programma
- per `y`, contenuta nel blocco del `main`
- per `x`, contenuta nel blocco del `for`
- per `i`, contenuta nel blocco del `for`.

Le associazioni nomi-oggetti contenute in un certo ambiente, relativo ad un certo punto del programma, all'interno di un blocco `B`, si distinguono in:

- *locali*, se la corrispondente dichiarazione è interna a `B`
- *non locali*, se la corrispondente dichiarazione non è interna a `B` (e quindi interna ad un blocco contenente `B`).

In particolare, le associazioni non locali relative a dichiarazioni contenute nel blocco del programma (il blocco più esterno) sono dette *globali*. I nomi introdotti da queste dichiarazioni (che diremo, appunto, *dichiarazioni globali*) sono visibili in tutto il programma, a meno che non vengano ridefiniti in blocchi più interni.

Con riferimento all'esempio di sopra, nell'ambiente all'interno del `for`, dopo la dichiarazione di `x`, abbiamo che:

- le associazioni relative a `x` e `i` sono locali
- l'associazione relativa a `y` è non locale
- le associazioni relative a `max` e `n` sono globali.

3.11 *Domande per il Capitolo 3*

1. Qual è il risultato prodotto dallesecuzione del seguente frammento di programma C++, con `x = 3` e `y = 5`? (attenzione alle parentesi ...)

```
if  (x>y) max = x;
          cout << Il maggiore e << x << endl;
if (x<=y) max = y;
          cout << Il maggiore e << y << endl;
```

2. Qual è il risultato prodotto dallesecuzione del seguente statement `if`, con `x = 5`?

```
if (x % 2)  cout << "then";
else cout << "else";
```

3. Dare il significato dello statement

```
do S; while(E);
```

   tramite un diagramma di flusso.

4. Dare il significato dello statement

```
for(int x=e1; x < e2; x++)S;
```

   (`e1`, `e2`: espressioni intere) tramite un diagramma di flusso.

5. È sempre possibile realizzare la struttura di controllo del `for` tramite `while`? E viceversa? Giustificare le risposte.

6. Citare almeno due motivi per cui è utile, in generale, introdurre in un linguaggio di programmazione altri costrutti oltre all'`if-else` ed al `while`.

7. Dare il significato dello statement

```
switch(e)
    {case c1: case c2: S1; break;
     case c3: S2; break;
     ...
     case cn: Sm; break;
     default: S;
    }
```

   (dove `e` è un'espressione di tipo t e `c1`, `c2`, ..., `cn` sono costanti di tipo t) tramite un diagramma di flusso.

8. Indicare vantaggi/svantaggi dello statement `switch` rispetto allo statement `if-else`.

9. È sempre possibile realizzare la struttura di controllo dello `switch` tramite `if-else` annidati? E viceversa? Giustificare le risposte.

10. Dare il significato dello statement

```
switch(e) {
   case c1: S1;
   case c2: S2;
   ...
   case cn: Sn;
   default: Sn+1;
}
```

(dove `e` è un'espressione di tipo t e `c1`, `c2`, ..., `cn` sono costanti di tipo t) tramite un diagramma di flusso (n.b.: gli statement `S1`, `S2`, ..., `Sn` non contengono lo statement `break`}).

11. Dare il significato dello statement

```
switch(e) {
   case c1: case c2: case c3: S1; break;
   case c4: case c5: S2;
}
```

(dove `e` è un'espressione di tipo t e `c1`, `c2`, ..., `c5` sono costanti di tipo t) tramite un diagramma di flusso.

12. Cosa prevede il Teorema di Böhm-Jacopini? Quali sono le sue implicazioni sui linguaggi di programmazione?
13. Qual è la forma sintattica ed il significato dello statement `goto` in C++? Cosa si intende con il termine programmazione strutturata?
14. A cosa servono in generale le "regole di scope" in un linguaggio di programmazione? A quali identificatori si riferiscono in generale?
15. Cosa si intende con il termine "blocco" in un linguaggio di programmazione? Quali dichiarazioni comprende in particolare il blocco di una funzione, quello dello statement `for`, e quello dellambiente globale in un programma C++?
16. Cosa affermano le "regole di scope" del C++?
17. Cosa si intende con ambiente di riferimento "locale" e "non locale"?
18. In base alle "regole di scope" del C++, cosa accade se in un blocco B_1 interno ad un blocco B_0 viene ridichiarata una variabile `x` già dichiarata nel blocco B_0? È possibile riferirsi alla `x` del blocco B_0 allinterno di B_1? E allesterno del blocco B_1?
19. Cosa si intende con il termine "programmazione strutturata"?
20. Che caratteristica devono avere le strutture di controllo in un linguaggio per la programmazione strutturata?
21. Quali vantaggi offre la programmazione strutturata?
22. Cosa si intende con il termine "codice spaghetti"?