

## Capitolo 5

# Costrutti base del linguaggio di programmazione (bozze, v. 1.0)

In questo capitolo illustreremo con maggiore dettaglio le operazioni elementari che il linguaggio di programmazione permette di esprimere e le modalità con cui tali operazioni possono essere composte in modo da formare un'algoritmo.

Per la corretta comprensione del capitolo devono essere conosciute le seguenti nozioni: dati di ingresso e di uscita di un algoritmo, sequenza statica e dinamica di un algoritmo, valore e attributo di un'informazione, rappresentazione e tipo di un'informazione.

### 5.1 Redazione di programmi

Per formulare un algoritmo in forma effettivamente eseguibile, occorre aggiungere alle istruzioni vere e proprie delle "istruzioni" particolari che servono, per esempio, a definire con precisione dove inizia e dove termina il programma, a specificare che il programma fa uso di particolari istruzioni di input e di output, ecc.

Prima di procedere a introdurre le istruzioni concrete che devono essere usate per programmare la macchina astratta è pertanto opportuno introdurre alcune di queste istruzioni aggiuntive in modo da essere subito in grado di scrivere programmi effettivamente utilizzabili. La descrizione che segue ha carattere essenzialmente pratico e prescrittivo. Una spiegazione completa della natura e del significato delle istruzioni introdotte verrà fornita successivamente.

Un programma completo consiste in una sequenza di istruzioni terminate dal carattere ; (punto e virgola), racchiusa tra parentesi graffe e preceduta dagli identificatori **int** `main`. In fondo alla sequenza di istruzioni deve poi essere aggiunta l'istruzione:

```
return 0;
```

anch'essa terminata dal carattere ;. Un programma ha pertanto la seguente forma generale:

```
int main () {  
    istruzione 1 ;  
    istruzione 2 ;  
    ...  
    istruzione n ;  
    return 0 ;  
}
```

Si noti la disposizione delle parentesi graffe e l'allineamento della sequenza di istruzioni leggermente sfalsate rispetto all'inizio della prima e dell'ultima riga. Questa disposizione del testo non è casuale. La disposizione più avanzata delle istruzioni viene detta *indentazione* e ha lo scopo di mettere in evidenza, anche dal punto di vista di chi legge il programma, quale sia la lista delle istruzioni e che esse sono interne al *blocco* delimitato dalle parentesi graffe. Anche l'allineamento verticale della parentesi graffa chiusa con l'inizio della prima riga ha lo scopo di aumentare l'impatto visivo della struttura del programma.

In tutti gli esempi riportati in questo testo, si farà largo uso dell'indentazione e di altri accorgimenti di tipo grafico per rendere sempre il più evidente possibile le relazioni tra le istruzioni e la struttura del programma. Sebbene tali accorgimenti siano di tipo convenzionale (il significato del programma non dipende da tali accorgimenti che possono essere omessi in tutto o in parte) e possano differire da programmatore a programmatore, tuttavia essi sono necessari per scrivere programmi facilmente leggibili da parte di operatori umani e non devono mai essere omessi.

Un'ultima aggiunta allo schema generale di programma appena illustrato, consiste nel premettere ad esso la frase:

```
# include <iostream.h>
```

tutte le volte (in pratica sempre nel caso dei semplici esempi che considereremo nel seguito) che il programma fa uso di istruzioni di input e/o di output.

## 5.2 Istruzione di output

L'istruzione di output ha la forma:

```
cout << espressione ;
```

dove *espressione* deve essere un'espressione ben formata.

L'effetto di un'istruzione di output è quello di generare il valore denotato dall'espressione, rappresentato secondo il tipo dell'espressione stessa, e di aggiungerlo allo standard output.

Spesso tale trasferimento viene indicato con il termine *stampa*<sup>1</sup> perchè in passato lo standard output era tipicamente associato ad un dispositivo di stampa su supporto cartaceo. Nei moderni sistemi di elaborazione lo standard output è invece normalmente associato ad una finestra del dispositivo grafico di visualizzazione, che consiste in un numero finito di righe (per es. 25), ciascuna contenenti un numero finito di caratteri (per es. 80).

### Osservazione 5.1

Con riferimento al concetto di uso di una variabile introdotto nel paragrafo 4.4, in una istruzione di output le variabili che compaiono nell'espressione vengono *usate*.

### Esempio 5.1

Il programma:

```
# include <iostream.h>

int main () {
    cout << 1254 + 36;
    return 0;
}
```

aggiunge allo standard output il valore 1290.

### Esempio 5.2

Il programma:

```
# include <iostream.h>

int main () {
    cout << (15.091 + 12.3) * 3;
    return 0;
}
```

aggiunge allo standard output il valore 82.173.

---

<sup>1</sup>Per esempio è tipica una frase del tipo "l'istruzione (di output) stampa il risultato".

**Esempio 5.3**

Il programma:

```
# include <iostream.h>

int main () {
    cout << 'a';
    return 0;
}
```

aggiunge allo standard output il valore corrispondente alla prima lettera minuscola dell'alfabeto inglese.

**Esempio 5.4**

Il programma:

```
# include <iostream.h>

int main () {
    cout << ``Paolo e' al mare``;
    return 0;
}
```

aggiunge allo standard output il valore corrispondente alla frase:

*Paolo e' al mare*

**Esempio 5.5**

Il programma:

```
# include <iostream.h>

int main () {
    cout << ``Prima riga``;
    cout << endl;
    cout << ``Seconda riga``;
    cout << endl;
    return 0;
}
```

aggiunge allo standard output il valore corrispondente alle frasi:

*Prima riga*  
*Seconda riga*

su due linee diverse, andando a capo dopo la seconda frase. L'andata a capo è prodotta aggiungendo allo standard output un carattere speciale di "fine linea", che si indica con l'identificatore `endl`.

## 5.3 Creazione di variabili

Come abbiamo detto nel paragrafo 4.4, le variabili devono essere esplicitamente create. Per farlo il linguaggio prevede apposite istruzioni che vengono dette *dichiarazioni*.

La dichiarazione di una variabile ha la forma generale:

*tipo variabile;*

Tipo	Insieme dei valori	rappresentazione
<b>bool</b>	valori logici appartenenti all'insieme $\{\mathbf{false}, \mathbf{true}\}$	<b>false</b> è rappresentato da una stringa nulla
<b>unsigned</b>	numeri naturali appartenenti all'intervallo $[0, N - 1]$ (valori tipici di $N$ : $2^{16} = 65536$ , $2^{32} = 4294967296$ )	rappresentazione posizionale
<b>int</b>	numeri relativi appartenenti all'intervallo $[-Z, Z - 1]$ (valori tipici di $Z$ : $2^{15} = 16384$ , $2^{31} = 2147483648$ )	rappresentazione in complemento a 2
<b>char</b>	caratteri comprendenti: cifre decimali, lettere maiuscole e minuscole dell'alfabeto inglese, segni di interpunzione e altri caratteri speciali per un totale di 256 simboli	rappresentazione ASCII
<b>float</b>	numeri reali appartenenti a un intervallo $[-2^{128}, -2^{128}]$	rappresentazione IEEE 754 (singola precisione)
<b>double</b>	numeri reali appartenenti a un intervallo $[-2^{1024}, -2^{1024}]$	rappresentazione IEEE 754 (doppia precisione)

Tabella 5.1: I principali tipi primitivi.

dove *tipo* è un identificatore che specifica un tipo primitivo o definito dall'utente, e *variabile* è un identificatore non usato in precedenza per altri scopi (questo vincolo non è in realtà necessario, ma per il momento assumeremo per semplicità che lo sia).

L'effetto della dichiarazione è quello che viene creata una cella di memoria associata al tipo e all'identificatore specificati.

Ad esempio, assumendo di disporre dei tipi primitivi elencati nella tabella 5.1, la dichiarazione:

```
int x;
```

aggiunge alla memoria una cella identificata dal nome *x* e in grado di contenere la rappresentazione di un intero relativo in complemento.

## 5.4 Istruzione di input

L'istruzione di *input* ha la forma:

```
cin >> variabile;
```

dove *variabile* deve essere l'identificatore di una variabile creata da una precedente dichiarazione.

L'effetto di un'istruzione di *input* è quello di estrarre il primo valore presente nello standard input e di memorizzarlo nella variabile specificata. Il valore viene memorizzato usando la rappresentazione corrispondente al tipo della variabile. Se il valore letto non è compreso nell'insieme dei valori corrispondente al tipo della variabile, il risultato della memorizzazione è indefinito. Se lo standard input è vuoto, l'istruzione di *input* rimane sospesa indefinitamente fino a che non viene aggiunto un valore allo standard input tramite la periferica collegata.

### Osservazione 5.2

Con riferimento al concetto di definizione di una variabile introdotto nel paragrafo 4.4, in una istruzione di *input* la variabile che compare in un'istruzione di *input* viene *definita*.

### Esempio 5.6

Il programma:

```
# include <iostream.h>

int main () {
    int x;
    cin >> x;
    cout >> 2 * x;
    return 0;
}
```

crea una variabile **x** di tipo **int**, estrae il primo valore dallo standard input e ne trasferisce la rappresentazione in complemento a 2 in **x**, aggiunge allo standard output il doppio del valore trasferito in **x**. Per esempio, se il primo valore presente sullo standard input è 123, allo standard output viene aggiunto il valore 246.

### Esempio 5.7

Il programma:

```
# include <iostream.h>

int main () {
    double x;
    double y;
    cin >> x;
    cin >> y;
    cout >> x * y;
    cout >> x / y;
    return 0;
}
```

crea due variabili di tipo **double**, estrae due valori dallo standard input e trasferisce la loro rappresentazione in formato IEEE 745 (doppia precisione) nelle due variabili, aggiunge allo standard output il prodotto dei due valori estratti seguito dal loro rapporto. Per esempio, se i valori presenti sullo standard input sono 5.24 e 2, allo standard output vengono aggiunti (nell'ordine) i valori 10.48 e 2.62.

### Esercizi proposti

1. Scrivere un programma che legge un numero dallo standard input e lo stampa sullo standard output preceduto dalla didascalia:

*Il numero letto è:*

2. Scrivere un programma che legge due numeri interi dallo standard input e stampa sullo standard output il quoto e il resto della divisione intera del primo numero per il secondo.

## 5.5 Istruzione di assegnazione

Abbiamo già spiegato come l'interprete sia in grado di generare nuovi valori calcolando il risultato di espressioni. Abbiamo anche visto come i valori generati dalla valutazione di espressioni possano essere trasferiti all'esterno dell'esecutore mediante le istruzioni di output.

Ci occupiamo ora di un altro tipo di istruzione che contiene espressioni: l'istruzione di *assegnazione*. Analogamente all'istruzione di output, anche l'istruzione di assegnazione comprende un'espressione, ma il suo effetto, oltre alla valutazione dell'espressione, è quello di trasferire il valore calcolato dall'interprete in una variabile in memoria.

La forma generale di un'istruzione di assegnazione è la seguente:

*variabile = espressione ;*

dove *variabile* rappresenta l'identificatore di una variabile creata da una dichiarazione precedente, il simbolo = rappresenta l'*operatore di assegnazione*, e *espressione* è un'espressione. Il tipo della variabile a sinistra dell'operatore di assegnazione e dell'espressione alla sua destra devono coincidere.

Quando l'esecutore esegue un'istruzione di assegnazione, l'interprete compie nell'ordine le seguenti operazioni:

- valuta l'espressione a destra dell'operatore di assegnazione;
- sulla base del tipo dell'espressione genera la rappresentazione del risultato ottenuto;
- trasferisce tale rappresentazione nella variabile a sinistra dell'operatore di assegnazione.

Per chiarire quanto detto presentiamo alcuni esempi. Se la variabile *x* è di tipo **int** l'istruzione:

```
x = 23;
```

valuta l'espressione di tipo **int** a destra dell'operatore di assegnazione e trasferisce la rappresentazione (in complemento) del valore ottenuto (il numero ventitrè) nella variabile *x*. Se la variabile *w* è di tipo **char** l'istruzione:

```
w = 'a';
```

valuta l'espressione di tipo **char** a destra dell'operatore di assegnazione e trasferisce la rappresentazione (ASCII) del valore ottenuto (la prima lettera minuscola) nella variabile *w*. Se le variabili *y* e *z* sono di tipo **int**, e il valore di *z* è 514, l'istruzione:

```
y = z;
```

valuta l'espressione di tipo **int** a destra dell'operatore di assegnazione e trasferisce la rappresentazione (in complemento) del valore ottenuto (il numero 514) nella variabile *y*. Se le variabili *alfa*, *x* e *y* sono di tipo **double**, il valore di *x* è 16.56, e il valore di *y* è -5.011, l'istruzione:

```
alfa = x + y;
```

valuta l'espressione di tipo **double** a destra dell'operatore di assegnazione e trasferisce la rappresentazione (in virgola mobile, doppia precisione) del valore ottenuto (il numero 11.549) nella variabile *alfa*. Infine, se la variabile *x* è di tipo **unsigned** e il suo valore è 10, l'istruzione:

```
x = x + 1;
```

valuta l'espressione di tipo **unsigned** a destra dell'operatore di assegnazione e trasferisce la rappresentazione (in rappresentazione posizionale) del valore ottenuto (il numero 11) nella stessa variabile *x*.

Nell'ultimo caso è evidente come, nel linguaggio dell'esecutore, il simbolo = non vada confuso con l'operatore di uguaglianza. L'operatore di assegnazione indica il trasferimento in memoria della rappresentazione di un valore, e dunque, scrivere  $x = x + 1$  indica semplicemente che:

- viene *usata* la variabile *x*;
- il suo valore (ad esempio 10) viene sommato a 1;
- la rappresentazione del risultato viene trasferita nella stessa variabile *x* *sovrascrivendo evidentemente il precedente valore contenuto in x*.

### Osservazione 5.3

Con riferimento ai concetti di definizione e uso di una variabile introdotti nel paragrafo 4.4, in una assegnazione le variabili che compaiono nell'espressione a destra dell'operatore di assegnazione vengono *usate*, mentre la variabile che compare a sinistra dell'operatore di assegnazione viene *definita*. Si noti che, come accade nell'ultimo esempio, una stessa variabile può essere usata e definita nella stessa istruzione di assegnazione. In questo caso la variabile viene prima usata e poi definita.

### Osservazione 5.4

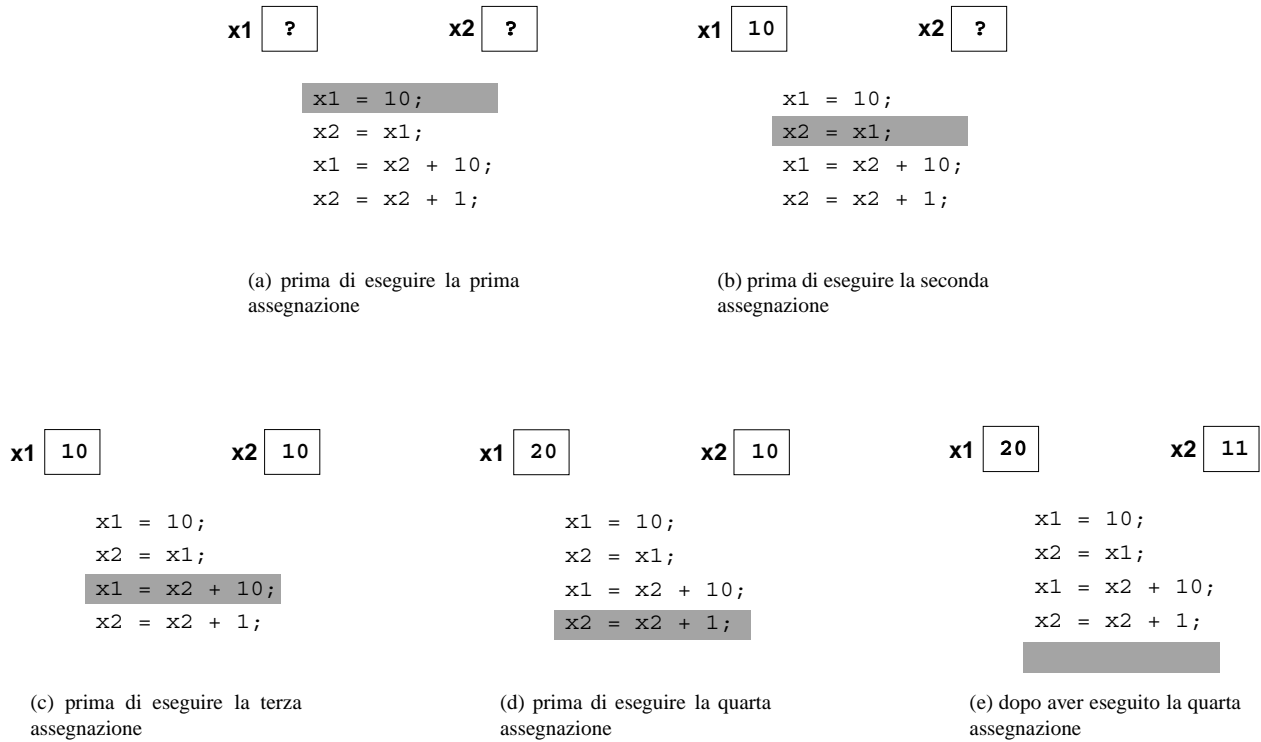


Figura 5.1: Effetti prodotti da una sequenza di assegnazioni.

Se il tipo dell'espressione non coincide con il tipo della variabile, la stringa di bit generata, che rappresenta il valore del risultato dell'espressione sulla base del tipo di quest'ultima, viene "convertita" in una stringa di bit che possa essere trasferito nella variabile (che sia cioè coerente con il suo tipo) prima di effettuare il trasferimento. La conversione può consistere in una trasformazione della stringa secondo regole previste per il caso specifico, può consistere nel troncatura la stringa o nell'estenderla nel caso fosse troppo lunga o troppo corta, o può consistere semplicemente nel trasferire la stringa senza effettuare modifiche.

Ad esempio, se il tipo dell'espressione fosse **int** e il tipo della variabile fosse **float**, la stringa prodotta, che corrisponde a un numero relativo rappresentato in complemento, viene trasformata in una stringa di opportuna lunghezza che rappresenta lo stesso numero in virgola mobile. Viceversa, se il tipo dell'espressione fosse **int** e il tipo della variabile fosse **unsigned**, la stringa prodotta viene trasferita senza modifiche. Si noti che questo significa che il numero calcolato e quello contenuto nella variabile dopo l'assegnazione sono uguali solo se il primo valore è positivo, altrimenti sono diversi.

**Esempio 5.8**

Per chiarire ulteriormente il significato dell'istruzione di assegnazione, supponendo che **x1** e **x2** siano due variabili di tipo **int** appena dichiarate (e cioè non ancora definite), consideriamo la seguente sequenza di assegnazioni:

```
x1 = 10;
x2 = x1;
x1 = x2 + 10;
x2 = x2 + 1;
```

In figura 5.1 viene mostrato graficamente l'effetto dell'esecuzione in sequenza delle quattro assegnazioni. Le variabili **x1** e **x2** sono rappresentate mediante dei rettangoli, all'interno dei quali è indicato il loro valore *prima* dell'esecuzione dell'istruzione evidenziata. Il punto interrogativo indicato nella figura 5.1-a all'interno dei due rettangoli sottolinea il fatto che le due variabili sono inizialmente indefinite.

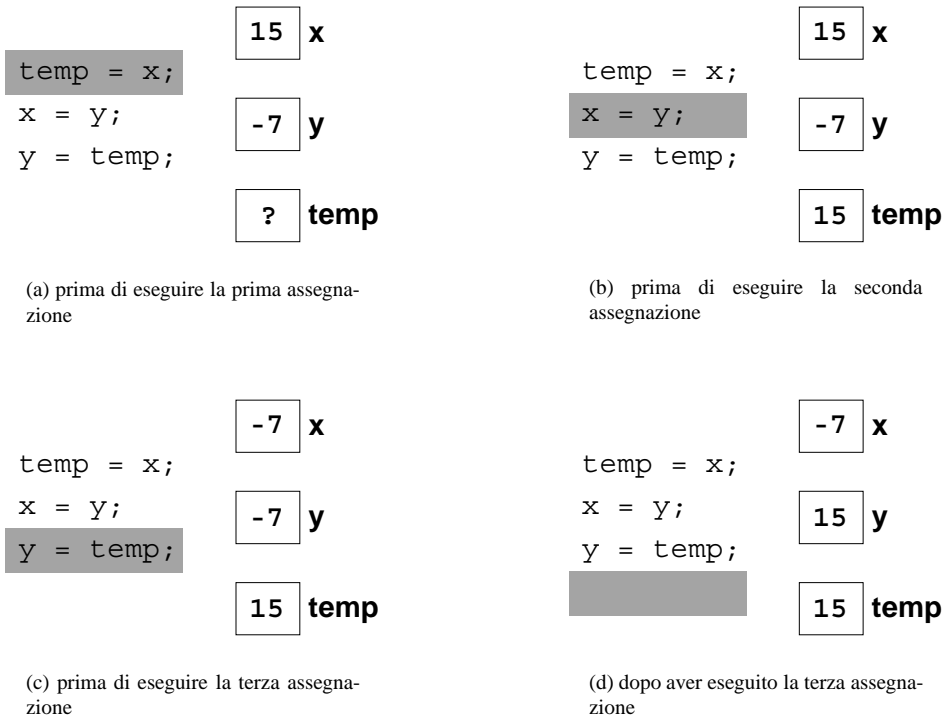


Figura 5.2: Scambio tra due variabili.

Si noti in particolare come una nuova assegnazione cancelli il valore precedentemente contenuto nella variabile che compare a sinistra dell'operatore di assegnazione.

**Esempio 5.9**

Un ulteriore esempio di uso dell'assegnazione è quello riportato nella figura 5.2 che mostra come risolvere il seguente problema tipico:

*date due variabili definite e dello stesso tipo scambiare i valori in esse contenuti.*

Il problema richiede l'impiego di una terza variabile di appoggio dedicata a questo scopo per poter "salvare" il valore di una delle due variabili mentre la si ridefinisce con il valore contenuto nell'altra. Ne deriva la seguente sequenza di assegnazioni in cui, si  $x$  e  $y$  sono le variabili di cui si vogliono scambiare i valori e  $temp$  è la variabile di appoggio:

```
temp = x;
x = y;
y = temp;
```

Si noti che inizialmente la variabile  $temp$  è indefinita (anche se fosse stata definita precedentemente, ai fini dello scambio il suo valore è irrilevante e dunque, da un punto di vista concettuale, va considerata indefinita). Analogamente il suo valore finale non è significativo perché dipende da quale delle due variabili da scambiare viene salvata in  $temp$ , cosa che, come abbiamo detto, è irrilevante (se nella sequenza di assegnazioni  $x$  e  $y$  si scambiano i ruoli il risultato finale è lo stesso).

Questo comportamento della variabile di appoggio  $temp$  è tipico come risulterà evidente quando discuteremo le proprietà dei segmenti di programma (cfr. capitolo 6).

**Esercizi proposti**

1. Riscrivere il programma dell'esempio 5.6, assegnando il risultato dell'espressione a una variabile `ris` e stampando il valore di tale variabile.
2. Riscrivere il programma dell'esempio 5.7, assegnando i risultati delle due espressioni a due variabili `ris1` e `ris2` e stampando i valori di tali variabili.
3. Riscrivere l'esercizio 2, usando una sola variabile `ris` a cui assegnare i risultati delle due espressioni prima di stamparli.
4. Usando 4 assegnazioni e una variabile di appoggio, scrivere una sequenza di istruzioni che, date tre variabili `x`, `y` e `z` dello stesso tipo, porti il valore di `y` in `x`, quello di `z` in `y`, e quello di `x` in `z`.

## 5.6 Blocco di istruzioni e variabili locali

Nel paragrafo 5.1 sono state introdotte in modo informale la notazione per indicare una sequenza di istruzioni e la nozione di *blocco* di istruzioni. Riprendendo l'argomento in modo sistematico, una sequenza di istruzioni viene indicata elencando le istruzioni da sinistra verso destra e dall'alto verso il basso, terminando ciascuna istruzione con il separatore `;`. Questa notazione indica che l'esecutore esegue una dopo l'altra le istruzioni della sequenza, nell'ordine con cui le istruzioni sono elencate. Ricordiamo che in questo caso la sequenza dinamica di ogni esecuzione coincide con la sequenza statica.

Un *blocco* è una sequenza di istruzioni (incluse eventuali dichiarazioni di variabili) racchiusa tra parentesi graffe. Un blocco di istruzioni può essere usato come un'unica *istruzione composta* per costruire istruzioni più complesse (vedi paragrafi successivi). Eventuali dichiarazioni di variabili all'interno di un blocco producono come sempre la loro creazione da parte dell'esecutore. Tali variabili, dette *locali* al blocco, vengono distrutte dopo che l'esecutore ha completato l'ultima istruzione del blocco. Le variabili locali possono pertanto essere definite o usate solo da istruzioni interne al blocco in cui sono state create.

### Esempio 5.10

Il seguente codice causa l'acquisizione di due valori interi dallo standard input e la stampa di due valori interi sullo standard output:

```
{
  int x;
  int y;
  cin >> x;
  {
    cin >> y;
    cout >> x+y;
  }
  cout >> x-y;
  cout >> 2*x;
}
```

Il codice è corretto perché tutte le variabili che le istruzioni definiscono e usano sono locali. Si noti che le variabili locali a un blocco sono locali anche ai blocchi interni ad esso.

Il seguente codice non è corretto perché la variabile `y`, locale al blocco più interno, non esiste al di fuori di tale blocco:

```

{
  int x;
  cin >> x;
  {
    int y;
    cin >> y;
    cout >> x+y;
  }
  cout >> x-y;
  cout >> 2*x;
}

```

Volendo mantenere la variabile *y* locale al blocco più interno bisognerebbe riscrivere il codice nel modo seguente:

```

{
  int x;
  cin >> x;
  {
    int y;
    cin >> y;
    cout >> x+y;
    cout >> x-y;
  }
  cout >> 2*x;
}

```

dove le istruzioni che usano *y* sono state spostate all'interno dello stesso blocco.

## 5.7 Controllo della sequenza dinamica

Nel capitolo 1 abbiamo visto che in un algoritmo sono normalmente presenti passi decisionali che servono a controllare la particolare sequenza dinamica che deve essere eseguita dall'esecutore per risolvere uno specifico caso di ingresso. Si può dimostrare che per descrivere qualunque algoritmo sono sufficienti solo due tipologie di passi decisionali a cui corrispondono nel linguaggio di programmazione due tipi di istruzioni: l'istruzione di *selezione* e l'istruzione *iterativa*. Si tratta in entrambi i casi di istruzioni composte da istruzioni più semplici che, sulla base del valore di verità di una condizione, consentono rispettivamente di scegliere tra due istruzioni alternative e di ripetere un'istruzione. Le due tipologie sono discusse nei dettagli nei due paragrafi successivi.

## 5.8 Istruzione di selezione

L'istruzione di selezione ha la seguente forma generale:

```

if ( condizione )
  parte-then;
else
  parte-else;

```

dove *condizione* è una condizione, mentre *parte-then* e *parte-else* sono due istruzioni qualsiasi (semplici o composte). L'esecuzione di un'istruzione di selezione produce nell'ordine:

1. la valutazione della condizione;
2. l'esecuzione della *parte-then* se il valore logico della condizione è **true** oppure l'esecuzione della *parte-else* se il valore logico della condizione è **false**.

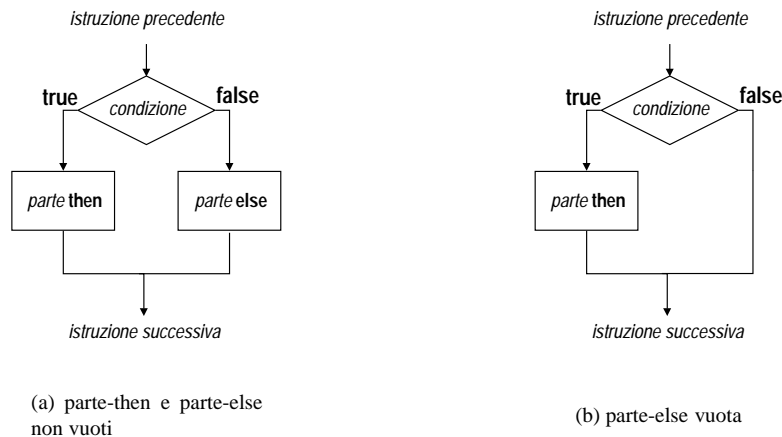


Figura 5.3: Diagramma di flusso dell'istruzione di selezione.

Si noti che l'istruzione di selezione produce l'esecuzione *solo* della parte-then oppure *solo* della parte-else.

### Esempio 5.11

Se le variabili  $a$  e  $b$  hanno lo stesso valore, l'istruzione:

```

if ( a == b )
    x = 0;
else
    x = 1;

```

causa prima la valutazione della condizione (  $a == b$  ), che risulta vera, e poi l'esecuzione dell'istruzione  $x = 0$ ; . Viceversa, se le variabili  $a$  e  $b$  avessero valore diverso, l'istruzione, dopo la valutazione della condizione, causerebbe l'esecuzione dell'istruzione  $x = 1$ ; .

### Osservazione 5.5

L'istruzione di selezione considerata corrisponde quindi alla seguente descrizione:

*se il valore contenuto nella variabile  $a$  è uguale a quello contenuto nella variabile  $b$  allora assegna il valore 0 alla variabile  $x$ , altrimenti assegna il valore 1 alla variabile  $x$*

Si noti che nella descrizione sono state evidenziate le parole:

- **se** che sottolinea il fatto che si opera una scelta; per questo motivo l'istruzione di selezione viene anche chiamata istruzione **if** (che significa "se" in inglese);
- **allora** che inizia la descrizione della parte-then ("then" in inglese significa "allora");
- **altrimenti** che inizia la descrizione della parte-else ("else" in inglese significa "altrimenti").

### Osservazione 5.6

Il comportamento delle istruzioni per il controllo della sequenza dinamica può essere descritto anche graficamente attraverso un *diagramma di flusso*, che è un diagramma in cui le condizioni sono rappresentate da rombi e le istruzioni sono rappresentate da rettangoli. I rombi e i rettangoli sono poi collegati da linee orientate per indicare l'ordine seguito dall'esecutore.

Il diagramma di flusso dell'istruzione di selezione considerata è rappresentato nella figura 5.3-a.

### Osservazione 5.7

La parte-then o la parte-else di un'istruzione di selezione può corrispondere all'*istruzione nulla*, cioè all'istruzione che non produce alcun effetto e che si indica con il separatore ;. In questo caso si dice che la parte-then o else è *vuota*.

Si può pertanto scrivere:

```
if ( a == b )
    ;
else
    x = 1;
```

che non produce alcun effetto se a e b hanno lo stesso valore, mentre causa l'assegnazione di 1 a x se a e b hanno valore diverso, oppure si può scrivere:

```
if ( a == b )
    x = 0;
else
    ;
```

che causa l'assegnazione di 0 a x se a e b hanno lo stesso valore, mentre non produce alcun effetto se a e b hanno valore diverso.

Se la parte-else coincide con l'istruzione nulla, la si può completamente omettere. Per esempio il secondo caso considerato potrebbe essere più semplicemente scritto come segue:

```
if ( a == b )
    x = 0;
```

Si noti però che in tutti questi casi non significa che non ci sia la parte-then o la parte-else, ma solo che la loro esecuzione non ha alcun effetto come indicato graficamente dal diagramma di flusso riportato in figura 5.3-b. L'osservazione è importante perché *nel progettare un algoritmo occorre considerare esplicitamente cosa accade quando viene eseguita una parte vuota di una istruzione di selezione*. Torneremo comunque sulla questione nel capitolo 6.

### Osservazione 5.8

La parte-then e la parte-else possono essere istruzioni semplici (come nell'esempio 5.11) o composte come nel seguente esempio:

```
if ( a == b ) {
    x = 0;
    y = 1;
}
else {
    x = 1;
    y = 0;
}
```

dove sia la parte-then che la parte-else sono blocchi contenenti una sequenza di due istruzioni. Se le variabili a e b hanno lo stesso valore, l'istruzione considerata causa l'assegnazione di 0 a x e di 1 a y, se hanno valore diverso vengono assegnati i valori invertiti.

### Osservazione 5.9

Si noti che se si omettono le parentesi graffe nella parte-else il significato cambia. Si consideri ad esempio, il codice:

```
if ( a == b ) {
    x = 0;
    y = 1;
}
else
    x = 1;
y = 0;
```

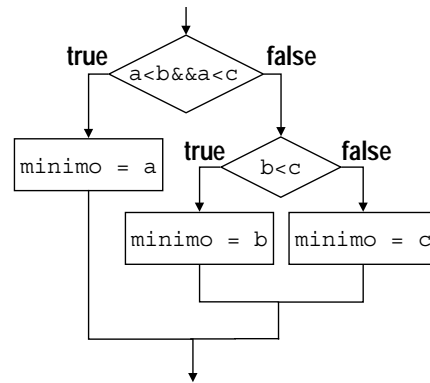


Figura 5.4: Diagramma di flusso dei istruzioni di selezione innestate.

L'istruzione di selezione causa l'assegnazione di 0 a  $x$  e di 1 a  $y$ , se le variabili  $a$  e  $b$  hanno lo stesso valore, mentre se hanno valore diverso causa solo l'assegnazione di 1 a  $x$ ; l'assegnazione di 0 a  $y$  è in sequenza all'istruzione di selezione e pertanto *viene eseguita dall'esecutore in ogni caso, indipendentemente dai valori di  $a$  e  $b$ .*

Si noti anche che se si omettono le parentesi graffe della parte-then si ottiene un codice scorretto senza significato.

### Osservazione 5.10

Si noti l'impiego dell'indentazione in tutti gli esempi di istruzione di selezione forniti. Essa viene sempre usata per mettere in evidenza *la struttura dell'istruzione* e cioè: la riga che contiene la condizione, le istruzioni che appartengono alla parte-then, le istruzioni che appartengono alla parte-else.

È tuttavia importante notare come l'indentazione è *solo una convenzione usata dai programmatori per migliorare la leggibilità dei programmi da parte di un lettore umano*. L'esecutore non ha in alcun modo bisogno dell'indentazione perché la presenza delle parole **if** ed **else**, unitamente all'uso corretto delle parentesi tonde e graffe, chiarisce in ogni caso il ruolo di ciascuna parte dell'istruzione. Anzi, è essenziale avere ben presente che sono queste cose a determinare *il significato del codice* e non l'indentazione utilizzata.

### Esempio 5.12

La parte-then e la parte-else possono a loro volta contenere istruzioni di selezione dando luogo a scelte in cascata che permettono di descrivere algoritmi che devono operare una scelta tra più di due alternative. Ad esempio, supponendo che le variabili  $a$ ,  $b$  e  $c$  di tipo **int** siano già state definite, per assegnare alla variabile  $\text{min}$ , anch'essa di tipo **int**, il valore minimo contenuto nelle tre variabili si può usare il codice:

```

if ( a < b && a < c )
    min = a;
else
    if ( b < c )
        min = b;
    else
        min = c;
  
```

che prevede una istruzione di selezione nella parte-else di un'altra istruzione di selezione. Per indicare questo tipo di composizione "a scatole cinesi" di istruzioni una dentro l'altra si usa il termine *annidamento* (dall'inglese *nesting*).

Applicando le regole illustrate fino ad ora è semplice comprendere il significato del codice, evidenziato anche dal diagramma di flusso della figura 5.4. Si noti che l'annidamento di due istruzioni **if** corrisponde a effettuare la scelta tra *tre* alternative. Non è difficile comprendere come usando ulteriori livelli di annidamento sia possibile operare scelte tra un qualunque numero di alternative.

**Esercizi proposti**

1. Scrivere un programma che legge tre numeri naturali che rappresentano i lati di un triangolo e stampa un messaggio che indica se il triangolo è equilatero, isoscele o scaleno (*suggerimento: usare uno degli algoritmi presentati nel paragrafo 2.6*).
2. Disegnare il diagramma di flusso corrispondente al seguente codice (la numerazione a sinistra non fa parte del codice):

```

1   min = a;
2   max = a;
3   if ( b < min )
4       min = b;
5   else
6       if ( b > max )
7           max = b;
8   if ( c < min )
9       min = c;
10  else
11  if ( c > max )
12      max = c;

```

3. Con riferimento al codice dell'esercizio 2, usando la numerazione delle istruzioni riportata sulla sinistra, indicare le sequenze dinamiche seguite dall'esecutore nei seguenti casi:
  - a vale 2, b vale 4, c vale 7;
  - a vale 7, b vale 4, c vale 2;
  - a vale 4, b vale 7, c vale 2;
4. Scrivere un programma che legge dallo standard input quattro numeri e stampa sullo standard output il valore minimo tra quelli letti.
5. Scrivere un programma che legge dallo standard input quattro numeri e stampa sullo standard output il valore minimo e il valore massimo tra quelli letti.
6. Scrivere un programma che legge dallo standard input due numeri e stampa sullo standard output i numeri letti in ordine crescente.
7. Scrivere un programma che legge dallo standard input tre numeri e stampa sullo standard output i numeri letti in ordine crescente.
8. Scrivere un programma che legge dallo standard input quattro numeri e stampa sullo standard output i numeri letti in ordine crescente.

**5.9 istruzione iterativa**

L'istruzione iterativa ha la seguente forma generale:

```

while ( condizione )
    corpo ;

```

dove *condizione* è una condizione, mentre *corpo* è una istruzione qualsiasi (semplice o composta).

l'esecuzione di un'istruzione iterativa produce nell'ordine:

1. la valutazione della condizione;
2. il termine dell'istruzione se il valore logico calcolato è **false** oppure l'esecuzione del corpo e la ripetizione del punto 1 se il valore logico calcolato è **true**.

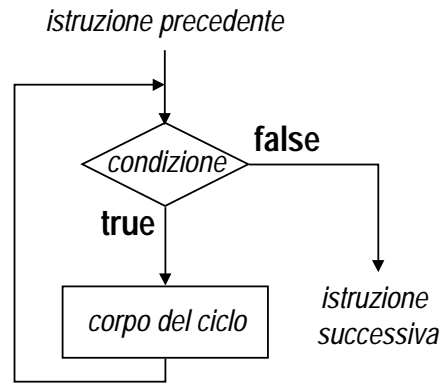


Figura 5.5: Diagramma di flusso dell'istruzione iterativa.

Si noti quindi che l'istruzione di selezione produce l'esecuzione del corpo *fintanto che il valore logico della condizione è true*. Si noti anche che se già la prima volta il valore logico della condizione è **false**, il corpo non viene eseguito neppure una volta. Il comportamento descritto giustifica il termine *condizione di uscita* normalmente usato per indicare la condizione dopo la parola **while** e il termine *ciclo* per indicare l'intera istruzione iterativa.

### Esempio 5.13

Se le variabili  $i$  e  $n$  contengono rispettivamente i valori 0 e 4, l'istruzione:

```

while ( i < n )
  i = i + 1;

```

causa prima la valutazione della condizione (  $i < n$  ) che risulta vera, poi l'esecuzione dell'istruzione  $i = i + 1$ ; che trasferisce in  $i$  il valore 1, poi di nuovo la valutazione della condizione (  $i < n$  ) che risulta vera, poi ancora l'esecuzione dell'istruzione  $i = i + 1$ ; che trasferisce in  $i$  il valore 2, e così via per altre due volte fino a quando la quarta esecuzione dell'istruzione  $i = i + 1$ ; trasferisce in  $i$  il valore 4; a questo punto viene nuovamente valutata la condizione (  $i < n$  ) che risulta falsa, causando la terminazione dell'istruzione iterativa.

Si noti che, se inizialmente le variabili  $i$  e  $n$  contessero rispettivamente i valori 3 e 4, il corpo dell'istruzione iterativa verrebbe eseguito solo una volta. Se invece contessero rispettivamente i valori 5 e 4, il corpo dell'istruzione iterativa non verrebbe eseguito affatto.

### Osservazione 5.11

L'istruzione iterativa considerata corrisponde quindi alla seguente descrizione:

**fintanto che** il valore contenuto nella variabile  $i$  è minore di quello contenuto nella variabile  $n$ , la variabile  $i$  viene incrementata di 1

Si noti che nella descrizione sono state evidenziate le parole **fintanto che** che sottolineano il fatto che si esegue ripetutamente una operazione; per questo motivo l'istruzione iterativa è anche chiamata istruzione **while** (che significa appunto "fintanto che" in inglese).

### Osservazione 5.12

Analogamente a quanto visto per la selezione, il diagramma di flusso dell'istruzione iterativa considerata è rappresentato nella figura 5.5.

### Osservazione 5.13

Il corpo di un ciclo può essere un'istruzione semplice (come nell'esempio 5.13) o composta come nel seguente esempio:

```

while ( i < n ) {
    i = i + 1;
    cout << i;
}

```

dove il corpo è un blocco contenente una sequenza di due istruzioni. Se inizialmente le variabili  $i$  e  $n$  valgono rispettivamente 0 e 4, l'istruzione iterativa considerata causa la stampa sullo standard output dei numeri: 1, 2, 3, 4.

#### Osservazione 5.14

Si noti che se si omettono le parentesi graffe che racchiudono il corpo il significato cambia. Si consideri ad esempio, il codice:

```

while ( i < n )
    i = i + 1;
cout << i;

```

Se inizialmente le variabili  $i$  e  $n$  valgono rispettivamente 0 e 4, l'istruzione iterativa ripete quattro volte *solo* l'incremento di  $i$  prima di terminare; l'istruzione di output è in sequenza all'istruzione iterativa e pertanto *viene eseguita dall'esecutore in ogni caso una sola volta*.

#### Osservazione 5.15

Si noti l'impiego dell'indentazione in tutti gli esempi di istruzione iterativa forniti. Essa viene sempre usata per mettere in evidenza *la struttura dell'istruzione* e cioè: la riga che contiene la condizione di uscita e le istruzioni che appartengono al corpo del ciclo.

Anche in questo caso l'indentazione è *solo una convenzione usata dai programmatori per migliorare la leggibilità dei programmi da parte di un lettore umano*. L'esecutore non ha in alcun modo bisogno dell'indentazione perché la presenza della parola **while**, unitamente all'uso corretto delle parentesi tonde e graffe, chiarisce in ogni caso il ruolo di ciascuna parte dell'istruzione.

#### Osservazione 5.16

Il corpo di un ciclo può contenere sia istruzioni di selezione, sia altri cicli dando luogo alla descrizione di algoritmi complessi. Anche nel caso dei cicli, per indicare questo tipo di composizione "a scatole cinesi" di istruzioni una dentro l'altra si usa il termine *annidamento*. In particolare si parla di *cicli annidati* (in inglese *nested*) per indicare la presenza di un ciclo nel corpo di un altro ciclo.

#### Osservazione 5.17

Un'ulteriore importante osservazione riguarda la terminazione di un'istruzione iterativa.

Poiché si assume in generale che un esecutore impiega un tempo finito per eseguire una qualunque istruzione semplice, è chiaro che l'esecutore impiega certamente un tempo finito per eseguire istruzioni composte formate solo da sequenze di assegnazioni e da istruzioni di selezione. Infatti in questo caso la lunghezza della sequenza dinamica è necessariamente finita e pertanto il tempo di esecuzione è una somma finita di termini finiti.

Questa conclusione non è però necessariamente vera allor quando l'esecutore deve eseguire un'istruzione iterativa. In questo caso la sequenza dinamica ha lunghezza finita *solo se dopo un numero finito di ripetizioni la condizione di uscita del ciclo diventa falsa*. In caso contrario l'esecutore, costretto a rispettare il significato dell'istruzione, non può che ripetere il corpo del ciclo indefinitamente e la corrispondente sequenza dinamica ha lunghezza infinita.

È evidente che tale comportamento non è di alcuna utilità pratica e che un algoritmo che ammettesse una tale sequenza dinamica tra quelle possibili conterrebbe un errore.

Purtroppo, stabilire se un ciclo non possa dar luogo a sequenze dinamiche di lunghezza infinita non è cosa in generale agevole. Con argomenti di buon senso è tuttavia possibile stabilire tale proprietà nella maggior parte dei casi più semplici.

#### Esempio 5.14

Con riferimento al codice riportato nell'esempio 5.13, il ciclo non può dar luogo a sequenze dinamiche di lunghezza infinita. Infatti, poiché ad ogni ripetizione il valore di  $i$  viene incrementato, qualunque siano i valori iniziali di  $i$  e  $n$

dopo un numero finito di ripetizioni il valore di  $i$  avrà raggiunto o superato quello di  $n$  (eventualmente zero ripetizioni se  $i$  fosse maggior o uguale a  $n$  fin dall'inizio).

### Esempio 5.15

Con riferimento al seguente codice:

```
while ( i != n ) {
    i = i + 2;
    cout << i-1;
    cout << i;
}
```

si osserva che il ciclo può dar luogo a sequenze dinamiche di lunghezza infinita. Infatti, se i valori iniziali di  $i$  e  $n$  fossero rispettivamente 0 e 3, a motivo degli incrementi ripetuti, la variabile  $i$  assumerebbe i valori: 0, 2, 4, ... e non diverrebbe pertanto mai uguale a  $n$ . Si tratta quindi di un codice errato. Nella pratica della programmazione è possibile che venga formulato per errore un ciclo che può non terminare. Si tratta di situazioni non sempre facili da rilevare nei casi concreti.

### Esercizi proposti

1. Scrivere un programma che stampa sullo standard output la tabella delle moltiplicazioni tra le cifre decimali.
2. Scrivere un programma che stampa sullo standard output i primi 10 numeri naturali.
3. Scrivere un programma che legge un numero naturale  $N$  e stampa sullo standard output i primi  $N$  numeri naturali.
4. Scrivere un programma che legge un numero naturale  $N$  e stampa sullo standard output le prime  $N$  potenze di 2.
5. Scrivere un programma che legge due numeri naturali  $b$  e  $N$  e stampa sullo standard output le prime  $N$  potenze di  $b$ .

## 5.10 Array

In tutti gli esempi visti finora, le informazioni da elaborare, quasi sempre di natura numerica, sono state caratterizzate da valori *atomici*, non scomponibili cioè in componenti più semplici. Esistono però nella pratica anche informazioni che per loro natura sono *strutturate*, i cui valori sono composti da valori più semplici, organizzati secondo particolari *strutture*.

Un tipico esempio di informazione strutturata è un numero complesso, formato da una coppia ordinata di numeri reali. La parte reale e la parte immaginaria sono gli elementi semplici componenti, mentre la struttura dell'informazione è quella di una *coppia ordinata*. Un altro esempio è quello di un vettore in uno spazio a  $n$  dimensioni, composto da una  $n$ -pla ordinata di valori numerici. Un terzo esempio è rappresentato da un insieme finito, composto da un numero finito di valori (non necessariamente numerici), la cui struttura questa volta non presuppone alcun ordinamento tra i componenti.

Per essere manipolate da un esecutore, le informazioni strutturate richiedono l'impiego di *variabili strutturate*, capaci cioè di contenere valori composti da elementi più semplici. Il numero di componenti e la struttura secondo cui sono organizzati i valori sono determinati dal *tipo* della variabile, che deve essere a sua volta un *tipo strutturato*.

Possono essere definiti diversi tipi strutturati, nessuno dei quali però è primitivo, cioè predefinito dal linguaggio. Il linguaggio, tuttavia, mette a disposizione dei *meccanismi di strutturazione* per definire tipi strutturati. Ciascun meccanismo determina le caratteristiche comuni ai tipi strutturati che possono essere definiti attraverso di esso. In particolare ciascun meccanismo determina se i componenti sono tutti dello stesso tipo (*componenti omogenei*) o no, e la cosiddetta *funzione di accesso*, cioè le modalità con cui si possono individuare le singole informazioni componenti a partire dall'informazione strutturata complessiva.

L'esempio di meccanismo di strutturazione di gran lunga più utilizzato nella programmazione è l'*array*, che permette di definire tipi strutturati con le seguenti caratteristiche:

- le informazioni componenti sono tutte dello stesso tipo  $T$ ;
- il numero  $N$  delle informazioni componenti è finito e determinato all'atto della scrittura del programma;
- la funzione di accesso è basata su una corrispondenza biunivoca tra le informazioni componenti e i primi  $N$  numeri naturali, e permette di individuare un singolo valore componente dell'array attraverso un numero naturale appartenente all'intervallo  $[0, N - 1]$ , che prende il nome di *indice*.

Queste caratteristiche sono essenzialmente quelle delle  $n$ -ple ordinate di valori omogenei che, come vedremo, trovano applicazione nelle più diverse situazioni.

Per dichiarare una variabile strutturata come array si può usare una dichiarazione del tipo:

```
T nome [EXT];
```

dove  $T$ , detto il *tipo base dell'array*, è un identificatore che indica il tipo dei singoli componenti, *nome* è l'identificatore associato alla variabile strutturata, ed *EXT* è l'*estensione* dell'array, cioè il numero (finito) delle componenti. La variabile strutturata, come tutte le variabili, ha un tipo associato che viene detto *array di tipo  $T$* <sup>2</sup>. Si noti che, nella dichiarazione di una variabile strutturata come array, *EXT* deve essere un'espressione *costante* di tipo intero. Si noti anche che, per quanto detto in generale sugli array, se indichiamo con  $n$  il valore dell'espressione costante intera *EXT*, gli  $n$  componenti dell'array sono individuati dagli indici da 0 a  $n - 1$ .

La forma usata nel linguaggio per indicare una particolare componente di una variabile strutturata come array è:

```
nome [ind];
```

dove *nome* è l'identificatore associato alla variabile, e *ind* è un'espressione *qualsiasi* di tipo intero. In particolare, *ind* può essere un'espressione non costante. La notazione appena illustrata corrisponde a indicare il componente della variabile strutturata che ha come indice il *valore* dell'espressione *ind*.

### Esempio 5.16

Volendo manipolare informazioni che rappresentano punti nello spazio tridimensionali, poiché ciascun punto è individuato da tre numeri reali, si può usare una variabile di tipo array di *double* con estensione 3, dichiarata nel modo seguente:

```
double punto3D[3];
```

Secondo quanto detto in precedenza, la variabile `punto3D` è composta da tre informazioni atomiche, ciascuna di tipo **double**, associate ai tre numeri naturali 0, 1, 2. Volendo leggere dallo standard input le coordinate di un punto nello spazio per conservarle nella variabile strutturata `punto3D` si potranno usare tre istruzioni di ingresso, una per ogni componente della variabile stessa:

```
cin >> punto3D[0];
cin >> punto3D[1];
cin >> punto3D[2];
```

Alternativamente, volendo evitare di usare tre istruzioni distinte per la lettura delle tre componenti, si può usare il codice:

```
i = 0;
while ( i < 3 ) {
    cin >> punto3D[i];
    i = i + 1;
}
```

### Osservazione 5.18

<sup>2</sup>Il tipo della variabile strutturata non è genericamente *array*, ma *array di tipo  $T$* , per sottolineare che l'array è solo un meccanismo di strutturazione che ha bisogno di un tipo base per generare un tipo strutturato.

Ciascun componente di una variabile strutturata come array può essere usato come se fosse una normale variabile del tipo base dell'array. In particolare, se il componente compare in un'espressione, il suo tipo e il suo contenuto vengono *usati* per determinare il tipo e il valore dell'espressione secondo le normali regole degli operatori coinvolti. Se invece il componente compare in una istruzione di input o a sinistra dell'operatore di assegnazione, il componente viene *definito*, viene cioè trasferito nella cella di memoria ad esso corrispondente il valore acquisito dallo standard input o calcolato sulla base dell'espressione a destra dell'assegnazione.

#### Osservazione 5.19

L'espressione usata per selezionare la componente di un array può essere non costante. In tal caso, le variabili presenti nell'espressione sono *usate* nell'istruzione in cui compare la componente. Si noti che questo è vero sia quando la componente è usata sia quando è definita. Ad esempio, nell'istruzione:

```
punto3D[j+1] = punto3D[i];
```

vengono usate le variabili  $j$ ,  $i$ , e la componente di `punto3D` associata all'indice  $i$ , mentre viene definita la componente di `punto3D` associata all'indice  $j+1$ .

#### Osservazione 5.20

Al momento dell'esecuzione di una istruzione in cui compare la componente di un array, l'espressione usata per selezionare tale componente deve avere un valore che corrisponda effettivamente a una componente. In altre parole, se l'array ha  $n$  componenti, l'espressione deve avere un valore compreso tra 0 e  $n-1$  inclusi. Con riferimento all'esempio riportato nell'osservazione 5.19, se `punto3D` ha tre componenti, il valore della variabile  $j$  deve essere compreso tra -1 e 1 inclusi, e il valore della variabile  $i$  deve essere compreso tra 0 e 2 compresi.

#### Osservazione 5.21

Il meccanismo di strutturazione come array è comune alla maggior parte dei linguaggi di programmazione. In particolare, in tutti i linguaggi che consentono la definizione di array, tali strutture sono omogenee e hanno i componenti associati a un indice. Tuttavia, da linguaggio a linguaggio possono variare aspetti secondari del meccanismo come ad esempio:

- la possibilità di usare espressioni variabili per indicare l'estensione;
- le limitazioni sulla natura degli indici;
- la possibilità di operare sull'intera variabile strutturata o su sue sottoparti diverse dai singoli componenti.

Come già visto, nel linguaggio che usiamo in questo contesto, rispetto a questi punti, gli array hanno le seguenti caratteristiche:

- l'estensione deve essere specificata mediante un'espressione costante;
- se  $n$  è l'estensione dell'array, gli indici che individuano le sue componenti sono sempre numeri naturali da 0 a  $n-1$ ;
- è possibile dichiarare l'array come variabile unica (sebbene strutturata), ma poi si può operare esclusivamente sulle sue componenti prese singolarmente (le elaborazioni su tutto o su parte dell'array richiedono l'uso esplicito di cicli).

## 5.11 Dichiarazione di costanti

Quando è richiesto di usare valori costanti all'interno di un programma (per esempio per indicare costanti universali nelle espressioni, l'estensione nelle dichiarazioni di array, ecc.) può essere conveniente usare un identificatore associato al valore costante invece di riportare esplicitamente il valore all'interno delle espressioni. Il vantaggio di questo modo di procedere è che se si vuole modificare il valore della costante (ad esempio perché si vuole aumentare il numero di cifre significative nel caso di costanti universali, o perché si vuole variare l'estensione di uno o più array) è

sufficiente modificare solo l'istruzione che associa l'identificatore alla costante invece di modificare una per una tutte le occorrenze del valore costante presente nel programma.

L'associazione di un identificatore a un valore costante ha una forma simile a una dichiarazione di variabile, con l'aggiunta della parola riservata **const** prima del tipo e del valore costante da associare preceduto dal simbolo =. Si noti che non è consentito modificare una costante così definita mediante un'assegnazione successiva.

### Esempio 5.17

Volendo definire una costante PIGRECO di tipo reale si deve usare la dichiarazione:

```
const double PIGRECO = 3.1415;
```

Ancora, volendo definire due array di interi di dimensione uno il doppio dell'altro si possono usare le dichiarazioni:

```
const int MAX_ELEMS = 100;
int a_singlo [MAX_ELEMS] ;
int a_doppio [2*MAX_ELEMS] ;
```

dove si noti che la seconda estensione è ancora un'espressione costante pur non essendo semplice.

### Osservazione 5.22

L'impiego del simbolo = nella dichiarazione di una costante prende il nome di *inizializzazione* e ha proprietà diverse rispetto all'assegnazione. Per questo motivo, la parte di dichiarazione che va dal simbolo = in poi viene detto *inizializzatore*

Si noti che gli inizializzatori possono essere usati anche nelle dichiarazioni di variabili. In tal caso la variabile viene creata già *definita* con il valore dell'inizializzatore. Ad esempio, la dichiarazione:

```
int i = 0;
```

crea una variabile associata all'identificatore **i** e al tipo **int**, già definita con il valore 0. Si noti che in questo caso, essendo **i** una variabile, il suo valore può essere successivamente modificato senza limitazioni.

Le differenze tra un inizializzatore e un'assegnazione non si possono cogliere pienamente nel contesto dei meccanismi linguistici illustrati fino a questo momento. Rinviando pertanto la spiegazione di tali differenze.

## 5.12 Operatori di autoincremento e autodecremento

Le operazioni di incremento e decremento di una variabile intera sono molto frequenti nella formulazione di algoritmi. Nel linguaggio sono stati pertanto introdotti due operatori che permettono di specificare l'incremento e il decremento di una variabile in modo sintetico.

L'*operatore di autoincremento* viene indicato con il simbolo ++ e, applicato a una variabile, la incrementa di uno. In altri termini, l'istruzione:

```
i++;
```

è del tutto equivalente all'assegnazione:

```
i = i + 1;
```

Analogamente, l'*operatore di autodecremento* viene indicato con il simbolo -- e, applicato a una variabile, la decrementa di 1. Anche in questo caso c'è la piena identità con l'assegnazione equivalente.

### Osservazione 5.23

Per quanto detto le istruzioni:

```
i++;
```

e:

```
i--;
```

sono equivalenti a delle assegnazioni che contengono la variabile *i* sia a destra che a sinistra dell'operatore di assegnazione. Di conseguenza tali istruzioni prima *usano* e poi *definiscono* la variabile riferita; di questo si deve tenere conto nell'applicazione della regola sull'uso delle variabili (cfr. paragrafo 4.4).

#### Osservazione 5.24

Gli operatori di autoincremento e di autodecremento possono essere indicati sia prima che dopo l'identificatore della variabile a cui devono essere applicati. Esiste una differenza di significato nei due casi, ma tale differenza non produce alcun effetto se l'autoincremento e l'autodecremento vengono usati esclusivamente come istruzioni isolate. Nel seguito ci limiteremo a tale uso, peraltro consigliabile in generale per motivi di leggibilità, e pertanto rimandiamo il lettore interessato ad approfondire la questione al manuale del linguaggio.

### 5.13 Ciclo a conteggio (for)

Nella formulazione di algoritmi è molto frequente il caso di dover far uso di un ciclo da ripetere un numero di volte noto *prima dell'inizio del ciclo*. In questi casi, si dovrebbe usare un codice del tipo:

```
i = 0;
while ( i < espressione ) {
    corpo
    i = i + 1;
}
```

dove *espressione* è un'espressione di tipo intero (anche non costante). Se le istruzioni rappresentate da *corpo* non modificano mai il valore di *i* e di *espressione*, il ciclo viene ripetuto un numero di volte esattamente pari al valore di *espressione*. Nel codice, la variabile *i* ha pertanto l'unico scopo di contare il numero di ripetizioni e viene pertanto detta *variabile di conteggio*.

Per semplificare la formulazione degli algoritmi, nel linguaggio è disponibile un'istruzione apposita per indicare un ciclo del tipo di quello appena mostrato, che prende il nome di *ciclo a conteggio* o *ciclo for* dal nome della parola riservata che lo caratterizza.

La forma del ciclo **for** è la seguente:

```
i = 0;
for ( i=0; i<espressione; i++ )
    corpo
```

e il suo significato è *esattamente lo stesso del ciclo while riportato sopra*. Si noti che le istruzioni rappresentate da *corpo* non devono modificare né il valore di *i* né quello di *espressione*<sup>3</sup>.

### 5.14 Commenti

Per migliorare la leggibilità dei programmi, il linguaggio prevede la possibilità di riportare nel codice dei *commenti*. Tali commenti *non fanno parte del programma* e sono completamente ignorati dall'esecutore. Il programmatore può pertanto utilizzarli per spiegare ad eventuali lettori umani aspetti del programma *non immediatamente evidenti dal codice*.

Esistono due forme di commenti. La prima, più generale, richiede che il testo del commento sia racchiuso tra i simboli:

```
/* ... */
```

<sup>3</sup>In realtà, poiché il ciclo **for** è solo una forma equivalente del ciclo **while**, nel linguaggio non è presente tale limitazione. Tuttavia noi la rispetteremo sempre volendo usare il ciclo **for** esclusivamente per i casi in cui il numero di ripetizione del ciclo sia noto prima dell'inizio del ciclo stesso.

Tali commenti possono essere posti in un punto qualunque del programma e possono occupare anche più righe di testo. La seconda forma di commento inizia con il simbolo:

```
// ...
```

e può essere posto solo nella parte finale di una linea di codice, in quanto è automaticamente terminato dalla fine della linea.

## 5.15 Impiego di un ambiente di programmazione

Dopo aver introdotto i costrutti base del linguaggio ed essere dunque in grado di scrivere semplici programmi completi, è opportuno descrivere brevemente come occorre procedere per giungere all'esecuzione di un programma su un esecutore reale.

Nel paragrafo 4.1 abbiamo accennato alla possibilità che l'esecutore reale comprenda il linguaggio e lo esegua direttamente e alla possibilità alternativa che il programma debba essere tradotto prima che possa essere eseguito.

Generalmente l'esecutore descritto in queste pagine richiede l'uso di un compilatore secondo il procedimento sintetizzato dalla figura 4.1-b. Tuttavia, per semplificare l'attività di programmazione, da diversi anni si sono diffusi degli *ambienti di sviluppo integrati* che, attraverso un'unica interfaccia verso il programmatore di tipo visuale, consentono di effettuare:

- la redazione del testo del programma (*editing*);
- la traduzione del programma in linguaggio macchina e la corrispondente generazione del programma eseguibile (*compilazione*);
- l'esecuzione del programma.

L'uso degli ambienti di sviluppo semplifica molto l'uso pratico di un linguaggio per formulare algoritmi. Tuttavia la presenza della fase di traduzione non può essere mascherata al programmatore perché durante la traduzione possono essere rilevati e comunicati dal compilatore eventuali errori commessi nella scrittura del programma sorgente. Tale possibilità produce messaggi di errore non sempre facilmente comprensibili per cui le modalità di funzionamento di un compilatore meritano qualche commento ulteriore.

La compilazione viene effettuata passando attraverso quattro fasi: tre di analisi del testo del programma finalizzate alla sua comprensione, e una fase finale di traduzione vera e propria. Le tre fasi di analisi prendono il nome di *analisi lessicale*, *analisi sintattica* e *analisi semantica*, rispettivamente.

L'analisi lessicale effettua un raggruppamento dei caratteri di cui è composto il testo con lo scopo di individuare i *simboli* (detti anche *token*) del linguaggio: le parole chiave (ad esempio: **int**, **if**, **while**, ecc.), le costanti (ad esempio: -5, 8.09, 'a', **true**, ecc.), gli identificatori (ad esempio: X1, dato, y, a, ecc.), gli operatori (ad esempio: +, \*, ++, &&, ecc.), i separatori (ad esempio: ;, [, {, ', ecc.). In questa fase vengono riconosciuti ed eliminati i commenti.

L'analisi sintattica riconosce il tipo di istruzione (ad esempio: dichiarazione, assegnazione, selezione, iterazione, ecc.) e verifica che l'istruzione sia costruita correttamente, seguendo cioè le regole *sintattiche* del linguaggio (ad esempio se tutte le parentesi aperte sono chiuse, se le istruzioni sono terminate dal punto e virgola, ecc.).

L'analisi semantica verifica se sono rispettate *alcune* regole più complesse di quelle sintattiche (ad esempio se la variabili sono state dichiarate) e assegna ulteriori proprietà alle istruzioni (ad esempio il tipo a un'espressione).

Le prime tre fasi, possono dar luogo a errori se il programma non rispetta le regole lessicali, sintattiche e semantiche del linguaggio. Le segnalazioni non sono sempre precise e spesso sono così sintetiche da risultare oscure o generiche. È pertanto necessario fare molta pratica per acquisire la scioltezza necessaria a risolvere i casi di errori più comuni.

Se vi sono errori nelle fasi di analisi, infatti, il compilatore non effettua la quarta fase della traduzione che genera il programma eseguibile. Tuttavia, una volta che l'analisi non rileva errori, la traduzione viene sempre effettuata senza ulteriori segnalazioni di errore. Si noti però che questo non significa che il programma non contenga errori perché la capacità del compilatore di rilevare errori è limitata. In particolare un programma lessicalmente e sintatticamente corretto e che soddisfa alcune regole semantiche basilari corrisponde sempre a un algoritmo eseguibile. Tale algoritmo tuttavia potrebbe non corrispondere affatto alle intenzioni del programmatore ed essere addirittura privo di senso!

```

/
file prova.cpp
esemplifica le componenti piu' comuni
di un programma consente di mostrare
errori lessicali, sinatattici e di
semantica statica
*/
#include <iostream.h>
#include <stdlib.h>
. . .
    
```

line	message
3	unterminated character constant

Figura 5.6: Esempio di errore lessicale e relativa segnalazione da parte del compilatore.

```

int main()
{
    int x, y // dichiarazione
    cin >> x;
    y = 2*x;
    . . .
}
    
```

line	message
14	parse error before '>'

Figura 5.7: Esempio di errore sintattico e relativa segnalazione da parte del compilatore.

```

int main()
{
    int x, y // dichiarazione
    . . .
    cout << x; // una variabile
    cout << x-3*y1; // espressione
    . . .
}
    
```

line	message
17	'y1' undeclared (first use this function)
17	(Each undeclared identifier is reported only once
17	for each function it appears in.)

Figura 5.8: Esempio di errore semantico e relativa segnalazione da parte del compilatore.

A conclusione del capitolo riportiamo alcuni esempi di programmi che contengono errori di tipo lessicale, sintattico e semantico con le corrispondenti segnalazioni di errore generati dall'ambiente di sviluppo DevC++, versione 4.0.

Gli esempi sono riportati nelle figure 5.6, 5.7 e 5.8. In ciascun caso è riportato il testo parziale di un programma contenente un errore evidenziato da un cerchio, e, in basso, le segnalazioni di errore riportate dall'ambiente DevC++, che si appoggia su un compilatore di pubblico dominio.

Nella figura 5.6, l'errore consiste nell'aver ommesso il carattere \* all'inizio di un commento. Il compilatore pertanto tenta di analizzare il testo del commento e segnala errore in quanto esso non contiene ovviamente codice corretto. Si noti come peraltro la segnalazione di errore non individui correttamente né la causa reale dell'errore, né la sua localizzazione (l'errore è nella linea 1 e non nella linea 3 come segnalato). Nella figura 5.7, l'errore consiste nell'aver ommesso il punto e virgola dopo una dichiarazione. Si noti come anche in questo caso la segnalazione di errore non individui correttamente la causa reale dell'errore. Infine, nella figura 5.7, l'errore consiste nell'aver sbagliato il nome di una variabile ( $y_1$  al posto di  $y$ ). In questo caso la segnalazione di errore è abbastanza chiara.