

## Capitolo 7

# Sottoprogrammi (bozze, v. 1.0)

Finché il compito che un programma deve svolgere si riduce ad una semplice trasformazione di una quantità di dati limitata, il programma può essere formulato come un unico blocco di istruzioni.

Normalmente però, il compito da svolgere è complesso da un punto di vista algoritmico e con riferimento alle capacità dell'esecutore. Tale complessità si traduce nel fatto che la strategia risolutiva deve prevedere lo svolgimento di più sottocompiti. In altre parole il problema è decomponibile in più sottoproblemi, in relazione l'uno con l'altro secondo una certa struttura.

Per facilitare il compito del programmatore è opportuno che anche il programma (cioè la formulazione dell'algoritmo nel linguaggio di programmazione utilizzato) rispecchi la struttura del problema e sia suddiviso in *sottoprogrammi*, cioè in unità ben definite, caratterizzate da una certa autonomia, che possono essere composte secondo la strategia risolutiva prescelta.

In questo capitolo vengono pertanto discussi la definizione e l'impiego dei sottoprogrammi, che rappresentano in tutti i linguaggi di programmazione un eccezionale meccanismo concettuale di astrazione e di strutturazione degli algoritmi.

### 7.1 Libreria standard

Cominciamo con il definire informalmente un *sottoprogramma* come un insieme di istruzioni eseguibili dal nostro esecutore che portano alla soluzione di un compito ben definito. In altre parole, un sottoprogramma *contiene la descrizione di un algoritmo per quel compito*.

I sottoprogrammi sono entità dotate di una certa autonomia e, oltre ad essere impiegati nella risoluzione di compiti complessi, possono in essere predisposti a priori per svolgere compiti di utilità generale (comuni cioè a problemi anche molto diversi).

In particolare, sono già disponibili un certo numero di sottoprogrammi che formano la cosiddetta *libreria standard* (dall'inglese *standard library*). I sottoprogrammi della libreria standard sono divisi in categorie a seconda della loro funzione e per ciascuna categoria è disponibile, oltre al codice del sottoprogramma già tradotto in linguaggi macchina, anche un *header file* (vedi capitolo 10). Per utilizzare i sottoprogrammi appartenenti a una certa categoria, il programmatore deve aggiungere all'inizio del proprio programma una direttiva del tipo:

```
# include <nome header file>
```

Le principali categorie della libreria standard i nomi dei corrispondenti header file sono riportati nella tabella 7.1.

### 7.2 Chiamata a sottoprogramma

Quando, ad un certo passo di un algoritmo occorre svolgere un sottocompito per cui è già disponibile un sottoprogramma, possiamo usare una *chiamata al sottoprogramma* per indicare all'esecutore di risolvere il sottocompito usando quel sottoprogramma.

Nome dell'header file	Categoria
<code>ctype.h</code>	sottoprogrammi di conversione e di test su caratteri
<code>math.h</code>	funzioni matematiche in doppia precisione
<code>stdio.h</code>	sottoprogrammi di I/O (linguaggio C)
<code>iostream.h</code>	sottoprogrammi di I/O (linguaggio C++)
<code>fstream.h</code>	sottoprogrammi di I/O su file (linguaggio C++)
<code>stdlib.h</code>	sottoprogrammi di utilità generale
<code>string.h</code>	sottoprogrammi per la manipolazione di stringhe di caratteri
<code>time.h</code>	sottoprogrammi per la manipolazione di informazioni temporali

Tabella 7.1: Principali categorie di sottoprogrammi della libreria standard con i corrispondenti header file.

Esistono due tipologie di sottoprogrammi: le *funzioni* e le *procedure* che differiscono per il modo con cui la corrispondente chiamata può essere inclusa in un programma.

La chiamata a un sottoprogramma, indipendentemente se esso sia una funzione o una procedura, ha la seguente forma generale:

*nome* ( *lista dei parametri effettivi* )

dove *nome* è un identificatore che individua il sottoprogramma da usare e *lista dei parametri* è una lista di *parametri effettivi* separati da virgole. Si noti che, come chiariremo meglio nel paragrafo 7.3, *un parametro effettivo pu ò essere un'espressione o una variabile*.

### 7.2.1 Chiamata a funzione

Le funzioni sono sottoprogrammi che *restituiscono un valore*, detto *risultato* della funzione. La chiamata a una funzione è pertanto *un'espressione* che denota il valore restituito e la sua esecuzione *causa la valutazione di tale espressione*.

Per questo motivo la chiamata a una funzione non è un'istruzione autonoma, ma *deve* comparire all'interno di una qualunque istruzione che preveda la presenza di un'espressione.

Ad esempio, poiché nella categoria delle funzioni matematiche è disponibile una funzione della libreria standard che calcola il logaritmo naturale, assumendo che  $x$  e  $y$  sono variabili di tipo **double**, se fosse richiesto di assegnare a  $y$  il logaritmo naturale dell'espressione  $x+3.1$ , si può scrivere l'assegnazione:

```
y = log(x+3.1);
```

dove `log(x+3.1)` è la chiamata al sottoprogramma di tipo funzione di nome `log` e l'espressione  $x+3.1$  tra parentesi è l'unico parametro effettivo richiesto dalla funzione. Si noti come, nell'assegnazione, la chiamata coincida con l'espressione prevista a destra dell'operatore di assegnazione.

#### Esempio 7.1

Una chiamata alla funzione `log` potrebbe anche comparire nella condizione di uscita di un ciclo **while** o come argomento di un'istruzione di output, come nel seguente codice ( $a$  è una variabile di tipo **double**):

```
while ( log(a) > 0 ) {
    a = a/2;
    cout << a;
    cout << log(a);
}
```

Si noti che in tutti i casi la chiamata svolge il ruolo di un'espressione il cui valore è il risultato della funzione.

#### Osservazione 7.1

Durante l'esecuzione del programma, quando l'interprete valuta l'espressione che contiene la chiamata, il sottoprogramma di tipo funzione viene eseguito e il valore restituito viene usato per determinare il valore dell'espressione.

Nell'esempio precedente quindi la funzione `log` viene chiamata ed eseguita ogni volta che viene valutata la condizione di uscita del ciclo `while` e ogni volta che viene eseguita l'ultima istruzione di output nel corpo del ciclo. In totale quindi le chiamate a `log` sono pari al doppio del numero di volte che viene eseguito il corpo del ciclo `while` più uno.

### 7.2.2 Chiamata a procedura

Contrariamente alle funzioni, le procedure sono sottoprogrammi che non restituiscono un valore. La chiamata a una procedura è pertanto un'istruzione autonoma che ha l'effetto di produrre l'esecuzione della procedura da parte dell'esecutore.

Ad esempio, nella categoria dei sottoprogrammi di utilità generale, è inclusa la procedura `exit` che provoca la terminazione immediata del programma prima che si raggiunga l'ultima istruzione, e che prevede un unico parametro intero che serve per comunicare all'esecutore il motivo della terminazione<sup>1</sup>.

La chiamata al sottoprogramma di tipo procedura `exit` si esprime con l'istruzione:

```
exit(0);
```

Si noti il punto e virgola finale, richiesto dal fatto che la chiamata è un'istruzione autonoma.

#### Osservazione 7.2

Un procedura non può essere chiamata come una funzione perchè la sua chiamata *non* è un'espressione. Pertanto l'istruzione:

```
x = exit(0);
```

è errata.

Viceversa una funzione può essere chiamata come una procedura, usando un'istruzione autonoma. Ad esempio, tra le funzioni già pronte previste dal linguaggio ce ne è una che permette di far eseguire all'esecutore un comando non previsto dal linguaggio, ma previsto dall'ambiente operativo che gestisce la macchina reale (cfr. capitolo 9). La funzione è denominata `system`, ha come unico parametro una stringa di caratteri che corrisponde al comando da eseguire, e ha come risultato un valore intero che indica se il comando è stato eseguito con successo o meno. La chiamata a `system` dovrebbe pertanto essere inserita in un'espressione, come ad esempio nell'istruzione:

```
stato = system("dir a:");
```

Tale istruzione causa l'esecuzione del comando `dir a:` da parte dell'ambiente operativo esterno (con conseguente visualizzazione del catalogo corrispondente al floppy disk inserito nel drive `a:`) e l'assegnazione alla variabile `stato` di un valore intero che indica se vi sono stati problemi nell'esecuzione del comando (in genere un risultato nullo indica che non vi sono stati problemi, mentre un risultato diverso da zero indica che si è verificata una anomalia durante l'esecuzione del comando).

Se, tuttavia, non interessasse in alcun modo il risultato della funzione, essa potrebbe essere chiamata con l'istruzione autonoma:

```
system("dir a:");
```

che causa ancora l'esecuzione del comando `dir a:`. Questa volta, tuttavia, il valore restituito dalla funzione viene perduto.

Si noti che la possibilità di chiamare una funzione come una procedura è utile solo in alcuni casi particolari e che in generale non ha senso farlo. Ad esempio, nel caso della funzione `log`, l'istruzione:

```
log(2.03);
```

pur corretta dal punto di vista linguistico, è del tutto inutile perché l'unico effetto della chiamata è il calcolo del logaritmo naturale di `2.03`, ma tale valore, non essendo assegnato a variabili o usato nella valutazione di espressioni più articolate, viene perduto.

<sup>1</sup>In genere il ruolo di tale parametro non è di interesse nello sviluppo di semplici programmi e pertanto si usa come parametro effettivo una qualsiasi costante intera (per es. 0).

## 7.3 Interfaccia di un sottoprogramma

Nel paragrafo precedente abbiamo introdotto la chiamata a un sottoprogramma, specificando che essa è formata dal nome del sottoprogramma seguito dalla lista dei parametri effettivi. Nel caso il sottoprogramma sia di tipo funzione, la sua chiamata è un'espressione che denota il valore che viene prodotto dall'esecuzione del sottoprogramma, e che va inclusa in un'istruzione che utilizzi tale valore.

È quindi evidente che per chiamare correttamente un sottoprogramma occorre conoscere oltre al suo nome:

- se è una funzione o una procedura
- il tipo del suo risultato nel caso si tratti di un sottoprogramma di tipo funzione;
- il numero e la natura dei suoi parametri;
- il compito che esso svolge.

L'insieme di queste informazioni rappresenta tutto ciò che occorre sapere per usare il sottoprogramma allo scopo di risolvere un problema più ampio. Si noti peraltro che tali informazioni definiscono completamente ciò che il sottoprogramma può fare e pertanto costituiscono anche tutte le informazioni necessarie per scrivere l'algoritmo racchiuso dal sottoprogramma.

Per questo motivo l'insieme di informazioni sopra elencate formano l'*interfaccia* del sottoprogramma, cioè la descrizione completa delle informazioni condivisa da chi deve utilizzare il sottoprogramma e da chi lo deve implementare. Si noti che l'interfaccia costituisce una sorta di *contratto* tra le due parti.

Da un lato infatti, chi utilizza il sottoprogramma deve rispettare quanto specificato dall'interfaccia perché altrimenti la chiamata può dar luogo a comportamenti imprevisti. Ad esempio, se l'interfaccia indica che un sottoprogramma è una funzione che calcola il logaritmo di un numero, la chiamata non può essere usata per ottenere la radice quadrata del numero. Analogamente se il sottoprogramma prevede due parametri di tipo **double**, esso non può essere chiamato con una stringa di caratteri come unico parametro.

Dall'altro lato, chi implementa il sottoprogramma deve usare un algoritmo che compia tutto quanto è previsto dall'interfaccia. In caso contrario, infatti, potrebbe accadere che una chiamata corretta (che cioè rispetta l'interfaccia) possa produrre un risultato errato.

### 7.3.1 Prototipi

Da un punto di vista pratico l'interfaccia di un sottoprogramma viene descritta attraverso due componenti:

- un *prototipo*, che è una dichiarazione prevista dal linguaggio e che include, nell'ordine, le seguenti informazioni:
  - se il sottoprogramma è una funzione o una procedura e, nel primo caso, quale è il tipo del valore restituito;
  - il nome del sottoprogramma;
  - il numero, il tipo dei parametri e se essi possono essere modificati dal sottoprogramma;
  - per ciascun parametro la *modalità di passaggio*, e cioè se il corrispondente parametro effettivo deve essere un'espressione o il nome di una variabile;
- una *descrizione* completa del compito che il sottoprogramma svolge, incluso il significato dei parametri e dell'eventuale valore restituito; tale descrizione, per poter descrivere con accuratezza *cosa* fa il sottoprogramma, è normalmente fornita attraverso un commento contenente frasi in linguaggio naturale o altra notazione non prevista dal linguaggio.

#### Esempio 7.2

Di seguito sono riportate le interfacce dei sottoprogrammi usati nel paragrafo 7.2:

```

double log ( double x );
/* restituisce il logaritmo naturale di x */

void exit ( int exitcode );
/* termina il programma e comunica all'ambiente operativo
   il valore di exitcode */

```

Si noti che:

- prima del commento contenente la descrizione di cosa fa il sottoprogramma è riportato il prototipo che comprende:
  - il tipo del risultato se si tratta di una funzione (come nel caso del sottoprogramma `log`), o la parola riservata **void** per indicare che si tratta di una procedura;
  - l'identificatore corrispondente al nome del sottoprogramma;
  - una lista di dichiarazioni di parametri *formali* separate da virgole (negli esempi c'è un solo parametro in entrambi i casi);
- il prototipo è una dichiarazione del linguaggio ed è pertanto normalmente terminata dal un punto e virgola;
- le dichiarazioni relative ai parametri formali assomigliano alle dichiarazioni di variabili;
- la descrizione nei commenti è sintetica e *fa esplicito riferimento* ai parametri, chiarendo in tal modo il loro ruolo nel compito svolto dal sottoprogramma.

### Osservazione 7.3

Un confronto tra i prototipi dei sottoprogrammi `log` e `exit` e le corrispondenti chiamate (cfr. paragrafo 7.2) indica che il prototipo di un sottoprogramma descrive *come* deve essere formulata la chiamata. In particolare, i parametri effettivi presenti nella chiamata *devono*:

- corrispondere in numero e tipo ai parametri formali presenti nel prototipo;
- essere espressioni o, alternativamente, nomi di variabili, a seconda della modalità di passaggio dei corrispondenti parametri formali (vedi sottoparagrafi successivi).

### Osservazione 7.4

Per chiamare un sottoprogramma, è necessario che in precedenza sia stato aggiunto al codice il relativo prototipo.

Nel caso dei sottoprogrammi della libreria standard, il prototipo viene aggiunto mediante la direttiva `include`, applicata all'header file corrispondente.

Nel caso di sottoprogrammi definiti dal programmatore, per evitare errori nella traduzione del programma è bene aggiungere in testa al codice i prototipi di tutti i sottoprogrammi chiamati (cfr. esempio 7.4).

## 7.3.2 Passaggio per valore

Si consideri la seguente interfaccia:

```

double pow ( double x, double y );
/* restituisce il risultato di x elevato alla y */

```

di una funzione della libreria standard.

Come nei prototipi dell'esempio 7.2, nella lista dei parametri formali, per ciascun parametro, è indicato il nome preceduto dal tipo. Questo modo di dichiarare i parametri formali indica che al momento della chiamata i parametri formali vengono associati ai *valori* dei corrispondenti parametri effettivi che devono essere pertanto *espressioni*.

Al termine dell'esecuzione del sottoprogramma l'associazione tra i parametri formali e il valore dei parametri effettivi viene persa, e in un'eventuale successiva chiamata allo stesso sottoprogramma è possibile stabilire una nuova associazione tra i parametri formali e i valori dei corrispondenti parametri effettivi.

**Esempio 7.3**

Il programma:

```
# include <iostream.h>

int main () {
    cout << pow(2.5,3.6);
    cout << pow(-4.0,1.2);
    return 0;
}
```

chiama due volte la funzione `pow` stampandone il risultato. Nella prima chiamata il primo parametro di `pow` viene associato al valore reale `2.5` mentre il secondo parametro viene associato al valore reale `3.6` e viene stampato il risultato di  $2.5^{3.6}$ . Nella seconda chiamata il primo parametro di `pow` viene associato al valore reale `-4.0` mentre il secondo parametro viene associato al valore reale `1.2` e viene stampato il risultato di  $-4.0^{1.2}$ .

**Osservazione 7.5**

Questo modo di procedere, che prende il nome di *passaggio per valore dei parametri di formali*, è adatto per quei parametri che rappresentano i valori che il sottoprogramma deve inizialmente ricevere per svolgere il suo compito.

Ad esempio, con riferimento ai prototipi di `log`, `exit` e `pow`, i valori associati ai parametri formali sono necessari ai tre sottoprogrammi per calcolare, rispettivamente, il logaritmo naturale di un numero reale, comunicare all'ambiente operativo il valore di `exitcode`, calcolare la potenza di due numeri reali.

**7.3.3 Passaggio per riferimento**

Consideriamo la seguente interfaccia:

```
void leggi_data ( int &giorno, int &mese, int &anno );
/* legge dallo standard input tre numeri e li mette rispettivamente
   in giorno, mese ed anno */
```

Questa volta, nella lista dei parametri formali, tra il tipo e il nome di ciascun parametro compare il simbolo `&`. Esso indica che al momento della chiamata i parametri formali vengono associati alle *celle di memoria* dei corrispondenti parametri effettivi che devono essere pertanto *nomi di variabili*.

In effetti, dalla descrizione riportata nel commento, si comprende facilmente come i tre parametri della procedura rappresentino dei *parametri di uscita*, e cioè parametri che *ricevono un valore* a seguito dell'esecuzione del sottoprogramma. In altre parole, i parametri effettivi associati ai tre parametri formali (che sono per quanto appena detto delle variabili) vengono *definiti* dal sottoprogramma in modo del tutto analogo a quanto avverrebbe se i valori letti dallo standard input venissero direttamente assegnati ad essi.

Come nel caso del passaggio per valore, al termine dell'esecuzione del sottoprogramma l'associazione tra i parametri formali e le celle di memoria dei parametri effettivi viene persa, e in un'eventuale successiva chiamata allo stesso sottoprogramma è possibile stabilire una nuova associazione tra i parametri formali e le celle di memoria dei corrispondenti parametri effettivi.

Si noti però che, a differenza da quanto accade nel passaggio per valore, la perdita dell'associazione tra parametri formali e parametri effettivi al termine dell'esecuzione del sottoprogramma non significa che vengono persi i valori assegnati dal sottoprogramma. perché essi vengono assegnati direttamente alle celle di memoria corrispondenti ai parametri effettivi, la cui esistenza è indipendente dal sottoprogramma

**Esempio 7.4**

Il programma:

```

#include <iostream.h>

void leggi_data ( int &giorno, int &mese, int &anno );

int main () {
    int giorno_nascita;
    int mese_nascita;
    int anno_nascita;
    int giorno_oggi;
    int mese_oggi;
    int anno_oggi;
    leggi_data(giorno_nascita,mese_nascita,anno_nascita);
    leggi_data(giorno_oggi,mese_oggi,anno_oggi);
    if ( giorno_nascita==giorno_oggi && mese_nascita==mese_oggi )
        cout << "Oggi e' il tuo compleanno!";
    else
        cout << "Oggi non e' il tuo compleanno!";
    cout << endl;
    return 0;
}

```

chiama due volte la procedura `leggi_data`. Nella prima chiamata vengono letti dallo standard input tre numeri interi che vengono memorizzati, rispettivamente, nelle variabili `giorno_nascita`, `mese_nascita` e `anno_nascita`. Nella seconda chiamata vengono letti dallo standard input altre tre numeri interi che vengono memorizzati, rispettivamente, nelle variabili `giorno_oggi`, `mese_oggi` e `anno_oggi`.

#### Osservazione 7.6

Questo comportamento è dovuto al modo di associazione specificato per i tre parametri formali mediante la presenza del simbolo `&`. Tale simbolo indica che l'associazione tra i parametri formali e i parametri effettivi è *per riferimento* che significa che il sottoprogramma riceve un riferimento ai parametri effettivi e che, durante l'esecuzione del sottoprogramma, l'uso e la definizione dei parametri formali corrispondono in realtà a a uso e definizione dei parametri effettivi. In altre parole, quando un parametro formale è associato (o passato) per riferimento, esso *diviene un sostituto del parametro effettivo*.

#### Osservazione 7.7

Si noti che un parametro effettivo associato per riferimento *non* può essere un'espressione perché un'espressione non può essere né usata, né definita. Tali azioni possono essere effettuate solo sulle variabili e quindi un parametro effettivo associato per riferimento *deve* essere necessariamente una variabile.

#### Osservazione 7.8

Il parametro effettivo associato a un parametro formale passato per riferimento può essere un singolo elemento di un array, perché tale elemento è a tutti gli effetti una variabile del tipo base dell'array.

Ad esempio, volendo leggere una lista di date dallo standard input, si potrebbero usare tre array di interi "paralleli": `giorno` che deve contenere le informazioni relative al giorno di ciascuna data, `mese` che deve contenere le informazioni relative al mese di ciascuna data, e `anno` che deve contenere le informazioni relative all'anno di ciascuna data.

Assumando che la lista di date sia terminata dal fine file, e usando il sottoprogramma `leggi_data`, il segmento di codice che legge la lista è il seguente:

```

n_date = 0;
leggi_data(giorno[0],mese[0],anno[0]);
while ( !cin.eof() ) {
    n++;
    leggi_data(giorno[n_date],mese[n_date],anno[n_date]);
}

```

dove `n_date` è una variabile intera che svolge il ruolo di riempimento dei tre array paralleli.

### 7.3.4 Parametri formali array

Consideriamo la seguente interfaccia:

```
int conta_zeri ( int l[], int n );
/* restituisce il numero di zeri contenuto negli elementi
   dell'array l di indici da 0 a n-1 inclusi */
```

La funzione `conta_zeri` ha due parametri, il primo dei quali corrisponde a un array di interi. L'esempio mostra che un parametro formale array viene dichiarato aggiungendo dopo il nome una coppia di parentesi quadre "vuote", cioè senza l'estensione.

#### Osservazione 7.9

Il motivo per cui nel caso di parametri formali array non viene specificata alcuna estensione è che, come abbiamo avuto modo di evidenziare illustrando le modalità di passaggio dei parametri, *un parametro formale non è una variabile autonoma*, ma un nome che assume significato solo quando viene associato al corrispondente parametro effettivo in una chiamata. Un parametro formale array ha perciò di volta in volta l'estensione del corrispondente parametro effettivo.

#### Osservazione 7.10

Con riferimento al prototipo della funzione `conta_zeri` riportato sopra, un esempio di chiamata potrebbe essere il seguente (la chiamata è nell'ultima assegnazione):

```
. . .
const int MAX = 100;
int lista[MAX];
int len;
int tot_zeri;
. . .
/* istruzioni per definire len e gli elementi di lista da 0 a len-1 */
. . .
tot_zeri = conta_zeri(lista, len);
```

dove nella chiamata, come primo parametro effettivo, corrispondente al parametro formale array, viene indicato semplicemente il nome `lista`, dichiarato in precedenza come array di 100 interi. Durante l'esecuzione della funzione il parametro formale `l` viene pertanto associato all'array `lista` e ha un'estensione di 100.

#### Osservazione 7.11

Poiché nella dichiarazione di un parametro array non è presente il simbolo `&`, il parametro effettivo viene associato al parametro formale *per valore*. Questo però non significa che al parametro formale vengono associati i valori di tutte le caselle dell'array, perché il parametro effettivo, che coincide con nome dell'array senza alcuna estensione e senza alcun indice, non indica *tutto l'array*, ma solo un *riferimento* all'array<sup>2</sup>. Analogamente, anche l'aggiunta della coppia di parentesi vuote nella dichiarazione del parametro formale indica che si tratta di un *riferimento* a un array. Pertanto, al momento della chiamata, il *valore* del parametro effettivo (che, ripetiamo, è un riferimento a un array) viene associato al corrispondente parametro formale. Durante la sua esecuzione, il sottoprogramma può poi manipolare i singoli elementi dell'array attraverso tale riferimento.

#### Esempio 7.5

Consideriamo la seguente interfaccia:

<sup>2</sup>Si ricordi che nel paragrafo 5.10 abbiamo accennato al fatto che non è consentito manipolare l'intero array all'interno del codice.

```
int leggi_lista ( int l[] );
/* legge dallo standard input un intero N seguito da N interi;
   restituisce il valore di N e mette gli N interi letti negli
   elementi di l di indice da 0 a N-1 inclusi */
```

La funzione `leggi_lista` ha un parametro formale `l` che, al momento della chiamata, deve essere associato al riferimento a un array di interi.

La chiamata alla funzione `leggi_lista` ha la forma:

```
n = leggi_lista(lista);
```

dove `n` è una variabile intera e `lista` è il riferimento a un array di interi. Si noti che, il meccanismo di passaggio dell'array è lo stesso usato nel caso discusso nell'osservazione 7.10, ma questa volta il sottoprogramma ha il compito di modificare gli elementi del parametro effettivo array. La cosa è possibile proprio perché, nel caso degli array, ciò che viene passato è unicamente un riferimento all'array e pertanto il sottoprogramma, tramite tale riferimento ha accesso direttamente al parametro effettivo i cui elementi possono pertanto essere sia *usati* che *definiti*.

### 7.3.5 Parametri di ingresso e parametri di uscita

Negli esempi visti fino ad ora abbiamo incontrato tre categorie di parametri formali:

- i parametri passati per valore che non sono array;
- i parametri passati per riferimento che non sono array;
- i parametri array, sempre passati per valore.

Nel primo caso si tratta di *parametri di ingresso*, cioè di parametri che rappresentano *valori* da cui il sottoprogramma deve partire per svolgere il proprio compito.

Nel secondo caso si tratta di *parametri di uscita*, cioè di parametri che rappresentano *variabili* che il sottoprogramma deve modificare durante lo svolgimento del proprio compito. Si noti peraltro che, nel caso i parametri siano *contemporaneamente di ingresso e di uscita*, è in ogni caso necessario usare il passaggio per riferimento perché altrimenti il sottoprogramma non potrebbe modificarli.

Nel terzo caso si tratta di parametri che possono essere sia di ingresso, di uscita o di ingresso/uscita. Nel caso di array infatti è solo possibile passare un riferimento al parametro effettivo e pertanto il sottoprogramma ha sempre diretto accesso agli elementi dell'array.

Occorre pertanto tenere concettualmente distinti il ruolo del parametro (di ingresso, di uscita, di ingresso/uscita), che è legato alla natura del compito che il sottoprogramma deve svolgere, dalla modalità di passaggio che non dipende esclusivamente dal ruolo del parametro, ma è influenzata anche da altri aspetti come ad esempio se si tratta o meno di un array.

Per migliorare la corrispondenza tra il ruolo dei parametri formali e la forma della loro dichiarazione è possibile ricorrere alla parola riservata **const**. La presenza di tale parola nella dichiarazione di un parametro formale non modifica il modo di passaggio (che dipende unicamente dalla presenza del simbolo `&`), ma indica che il sottoprogramma *non può modificare* il valore del parametro, *qualunque sia la modalità di passaggio usata*.

È quindi possibili usare la parola riservata **const** per distinguere i parametri di ingresso da quelli uscita o di ingresso/uscita secondo le regole riportate nella tabella 7.2.

#### Esempio 7.6

Il sottoprogramma `conta_zeri` ha bisogno di ricevere il valore associato a `n` e di poter usare gli elementi di `l` di indice da 0 a `n-1` per contare il numero di zeri presenti. Pertanto, volendo usare la parola riservata **const** per evidenziare che entrambi i parametri sono di ingresso, il prototipo andrebbe riscritto come segue:

```
int conta_zeri ( const int l[], const int n );
```

Si noti ancora una volta che entrambi i parametri del sottoprogramma `conta_zeri` sono passati per valore e che:

Ruolo del parametro	Array	Forma della dichiarazione
ingresso	no	<b>const</b> <i>tipo nome</i>
uscita o ingresso/uscita	no	<i>tipo nome</i>
ingresso	sì	<b>const</b> <i>tipo_base nome</i> []
uscita o ingresso/uscita	sì	<i>tipo_base nome</i> []

Tabella 7.2: Convenzioni per la dichiarazione dei parametri formali in base al loro ruolo.

- nel caso del primo parametro, il nome formale *l* viene associato al valore del riferimento di un array, e tramite tale riferimento il sottoprogramma può usare gli elementi dall'array (cioè del parametro effettivo);
- nel caso del secondo parametro, il nome formale *n* è associato al valore del parametro effettivo (un'espressione di tipo intero), e il sottoprogramma può usare tale valore per svolgere il proprio compito.

### Esempio 7.7

Si consideri la procedura `leggi_lista2` con la seguente interfaccia:

```
void leggi_lista2 ( int l[], bf int &n );
/* legge dallo standard input un intero N seguito da N interi;
   assegna il valore di N a n e mette gli N interi letti negli
   elementi di l di indice da 0 a N-1 inclusi */
```

La procedura è una variante della funzione `leggi_lista` che restituisce la lunghezza della lista in un parametro *n*. Pertanto, entrambi i parametri sono di uscita come evidenziato dall'assenza della parola riservata **const** prima del tipo base di *l*, e dalla presenza del simbolo & (passaggio per riferimento) prima del nome *n*.

Si noti che questa volta i parametri del sottoprogramma `conta_zeri` sono passati uno per valore e uno per riferimento, e in particolare:

- nel caso del primo parametro, il nome formale *l* viene associato al valore del riferimento di un array, e tramite tale riferimento il sottoprogramma può definire gli elementi dall'array (cioè del parametro effettivo);
- nel caso del secondo parametro, il nome formale *n* è un sostituto del parametro effettivo, che deve necessariamente essere una variabile di tipo intero; il sottoprogramma può definire tale variabile durante lo svolgimento del proprio compito.

## 7.4 Implementazione di un sottoprogramma

Oltre a usare sottoprogrammi già pronti mediante l'istruzione di chiamata, il programmatore può *definire* nuovi sottoprogrammi.

Le modalità di tale definizione sono molto semplici in quanto un sottoprogramma non è altro che la descrizione di un algoritmo in una forma autonoma, distinta e parzialmente indipendente dal resto del programma che lo chiama. Un sottoprogramma si definisce pertanto descrivendo l'algoritmo che esso deve utilizzare mediante i normali costrutti del linguaggio di programmazione, e racchiudendo le istruzioni in un blocco associato al sottoprogramma.

Formalmente, la definizione di un sottoprogramma è costituita dal suo prototipo, detto *intestazione* (in inglese *header*, seguito da un blocco di istruzioni, detto *corpo* (in inglese *body*). A differenza della dichiarazione isolata di un prototipo introdotta nei paragrafi precedenti, nella definizione di un sottoprogramma l'intestazione non è terminata da un punto e virgola, ma è immediatamente seguita dalla parentesi graffa che inizia il corpo.

### Esempio 7.8

La definizione del sottoprogramma `leggi_data` è la seguente:

```
int leggi_data ( int &giorno, int & mese, int & mese ) {  
    cin >> giorno;  
    cin >> mese;  
    cin >> anno;  
}
```

**Osservazione 7.12**

Il corpo di un sottoprogramma è un normale segmento di programma. La particolarità è che, nel caso del corpo di un sottoprogramma, le variabili di ingresso coincidono con i parametri di ingresso, quelle di uscite coincidono con i parametri di uscita, e quelle di ingresso/uscita coincidono con i parametri di ingresso/uscita.

Nel corpo del sottoprogramma i parametri non devono essere dichiarati in quanto già compaiono nell'intestazione. Eventuali altre variabili che devono essere definite e usate nel corpo del sottoprogramma devono essere normalmente dichiarate all'inizio del blocco.