

## Capitolo 8

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

### 8.1 Stringhe di caratteri

I testi che può essere necessario trattare all'interno di un programma (nomi di file, didascalie, intestazioni, ecc.) sono in tutti i casi costituite da liste di caratteri. Aggiungendo caratteri speciali addizionali a quelli usuali è possibile rappresentare come caratteri anche aspetti grafici tipici dei testi come la suddivisione in linee (il corrispondente carattere speciale è denominato *newline* e indicato in C con il simbolo `\n`), l'allineamento a sinistra di colonne di testo (il corrispondente carattere speciale è denominato *tab* e indicato in C con il simbolo `\t`), ecc.

Quando si parla di liste di caratteri è usuale in informatica usare il termine *stringa*, sinonimo di lista o sequenza. Useremo pertanto il termine *stringa di caratteri*, o anche semplicemente *stringa*, per indicare il tipo di dato usato per rappresentare le informazioni testuali. Tali informazioni possono consistere in lunghi testi, in singole parole, anche di un solo carattere, o addirittura in un testo vuoto, cioè in una stringa di lunghezza nulla che denominiamo *stringa vuota*.

#### 8.1.1 Rappresentazione di stringhe di caratteri in C

In C le stringhe di caratteri sono rappresentate mediante *array e informazione tappo*. Una variabile di tipo *stringa di caratteri* è pertanto un normale array che ha per tipo base il tipo **char**. Questo significa che una variabile di tipo *stringa di caratteri* può contenere solo stringhe di lunghezza pari al numero di caselle dell'array meno uno (una casella dell'array deve essere sempre riservata per contenere l'informazione tappo).

Poiché i caratteri sono codificati attraverso il codice ASCII, le combinazioni di bit effettivamente utilizzate sono minori di quelle potenzialmente disponibili. Questo permette di individuare facilmente una combinazione di bit che possa essere usata come informazione tappo. La scelta è quella di usare la combinazione costituita da 8 zeri come informazione tappo. Per uniformità di terminologia, tale combinazione viene associata a un particolare carattere speciale denominato *carattere nullo* e indicato in C con il simbolo `\0`.

I meccanismi messi a disposizione dal linguaggio e dai sottoprogrammi di libreria per manipolare le stringhe di caratteri all'interno dell'esecutore presuppongono:

- che tutte le stringhe siano costituite da array di caratteri di lunghezza sufficiente a contenere la stringa da rappresentare;
- che subito dopo la casella contenente l'ultimo carattere della stringa esista almeno un'ulteriore casella contenente il carattere nullo.

Tali condizioni sono a carico del programmatore (o dell'utente nel caso delle operazioni di input) e se non sono soddisfatte la manipolazione delle stringhe da parte dell'esecutore può dare risultati non prevedibili e i corrispondenti programmi risultano non corretti.

#### Esempio 8.1

I seguenti sono esempi di dichiarazione di variabili di tipo stringa di caratteri:

```
const int MAX_LEN1 = 10;
char parola_breve [MAX_LEN1+1];

const int MAX_LEN2 = 100;
char linea [MAX_LEN2+1]

const int MAX_LEN3 = 10000;
char testo [MAX_LEN3+1];
```

Si noti che nel dichiarare l'estensione degli array, il termine 1 sommato alla costante intera assicura che la lunghezza massima ammessa per una stringa sia proprio pari al valore della costante simbolica. Si tratta di una convenzione di dichiarazione che riduce il rischio di dimenticare che la lunghezza massima consentita per la stampa è pari all'estensione dell'array meno uno.

### 8.1.2 Costanti

Le costanti di tipo stringa di caratteri si indicano scrivendo la stringa tra apici (il carattere ). Le costanti di tipo stringa sono rappresentate nella memoria dell'esecutore allo stesso modo dei valori memorizzati nelle variabili, e cioè come array di caratteri contenente i caratteri della stringa terminati dal carattere nullo. L'array di caratteri usato per rappresentare le costanti non ha un nome (non deve neppure essere dichiarato), ha lunghezza pari alla lunghezza della stringa costante che esso contiene più uno, per memorizzare il carattere nullo finale.

Si noti la differenza tra la notazione 'a' e "a". Nel primo caso la notazione indica il singolo carattere "a minuscola" che occupa un byte in memoria. Nel secondo caso la notazione indica una stringa, costituita dal solo carattere "a minuscolo" seguito dal carattere nullo, che occupa due byte. Come vedremo la seconda notazione può essere usata come parametro effettivo di un sottoprogramma che prevede un parametro di tipo stringa, mentre la prima notazione no.

### 8.1.3 Lettura di stringhe

La lettura di una parola limitata da spazi o da newline si effettua tramite l'istruzione `cin` in modo analogo agli altri tipi di dato. Assumendo di aver già scritto le dichiarazioni dell'esempio 8.1, l'istruzione:

```
cin >> parola_breve;
```

consuma e trasferisce i caratteri diversi da uno spazio nell'array `parola_breve` fino a che non si incontra uno spazio. Eventuali spazi iniziali presenti sullo standard input vengono consumati senza essere trasferiti nell'array. Lo spazio che fa terminare l'operazione di input non viene consumato. Si noti che, nell'esempio considerato, la lunghezza della parola letta (cioè della sequenza di caratteri consecutivi diversi da uno spazio che viene trasferita nell'array) deve essere minore o uguale a 10 (cioè al valore di `MAX_LEN1`). Come abbiamo già osservato, la verifica corrispondente è a carico dell'utente che immette i dati e non dell'esecutore.

#### Esempio 8.2

Come anche in altri esempi, per descrivere lo standard input, elencheremo nel seguito la stringa di caratteri da cui è costituito, indicando esplicitamente la presenza di spazi, newline e tab con i simboli `\b`, `\n` e `\t`, rispettivamente, e separando graficamente i caratteri con spazi (che non fanno pertanto parte dell'input).

Ciò premesso, se lo standard input contiene i caratteri:

```
Q u e s t o \b è \b u n \b t e s t o
```

l'istruzione:

```
cin >> parola_breve;
```

consuma i primi sei caratteri dello standard input e li memorizza nelle prime sei celle (dall'indice 0 all'indice 5 inclusi) dell'array di caratteri `parola_breve`. L'istruzione memorizza inoltre il carattere nullo nella settima cella dell'array. Dopo tale istruzione la variabile `parola_breve` contiene pertanto la stringa `Questo`.

Se la precedente istruzione viene eseguita una seconda volta, la seconda esecuzione consuma senza trasferirlo in memoria il carattere spazio che segue la parola `Questo`, consuma quindi il carattere `è` memorizzandolo nella prima cella dell'array `parola_breve` (sovrascrivendo il carattere precedentemente memorizzato), e memorizza il carattere nullo nella seconda cella dell'array. Dopo tale seconda istruzione di input la variabile `parola_breve` contiene pertanto la stringa `è`.

### 8.1.4 Sottoprogrammi di libreria per la manipolazione di stringhe

La maggior parte delle elaborazioni sulle stringhe di caratteri possono essere effettuate usando un insieme di sottoprogrammi inclusi nella libreria standard che operano su array di caratteri contenenti stringhe terminate dal carattere nullo. Vengono riportati di seguito i sottoprogrammi di uso più comune utilizzati in tutti gli esempi successivi. Alcuni dei seguenti prototipi sono leggermente modificati rispetto alla loro formulazione standard reperibile sui manuali per semplificarne l'uso da parte degli allievi. In tutti gli esempi presentati di seguito si può fare riferimento al prototipo qui presentato.

```
void strcat ( char s1[], char s2[] );
/* concatena la stringa s2 dopo la stringa s1 e pone la stringa
   risultante in s1 */

void strcpy ( char s1[], char s2[] );
/* assegna la stringa s2 alla stringa s1 */

int strlen ( char s[] );
/* restituisce la lunghezza della stringa s (escluso il carattere nullo
   finale) */

bool strcmp ( char s1[], char s2[] );
/* restituisce true se s1 è diversa da s2, restituisce false se s1 è
   uguale a s2 */
```

#### Esempio 8.3

Date le dichiarazioni:

```
const int MAX_TESTO = 10000;
const int MAX_PAROLA = 100;
char parola[MAX_PAROLA+1];
char testo[MAX_TESTO+1];
```

Supponendo di voler leggere nella variabile `testo` un testo costituito da parole separate da spazi il cui numero venga fornito subito prima del testo stesso, si può usare l'algoritmo:

```
int n, i;
cin >> n;
strcpy(testo, "");
for ( i=0; i<n; i++ ) {
    cin >> parola;
    strcat(testo, parola);
    strcat(testo, " ");
}
```

Si noti che dopo aver concatenato una nuova parola alla parte di testo già letta, memorizzata nella variabile `testo`, viene concatenata una stringa costante costituita da uno spazio per evitare che parole del testo risultino tutte attaccate.

Si noti anche come il segmento di programma riportato sopra segue lo schema di scansione con accumulatore di una lista di lunghezza nota. La cosa risulta evidente confrontando il precedente segmento con il segmento:

```
int n, i, somma;
cin >> n;
somma = 0;
for ( i=0; i<n; i++ ) {
    cin >> x;
    somma = somma + x;
}
```

Nel segmento che legge il testo, l'istruzione:

```
strcpy(testo, "");
```

ha l'effetto di assegnare la stringa vuota alla variabile `testo`. Tale istruzione equivale pertanto a un assegnazione che non può essere espressa col la notazione normale:

```
testo = "";
```

perché tale notazione non ha senso in C in quanto, secondo le regole del linguaggio, l'identificatore `testo` rappresenta un riferimento all'array e non l'array vero e proprio.

Analogamente, nel segmento che legge il testo, le istruzioni:

```
strcat(testo, parola);
strcat(testo, " ");
```

hanno l'effetto di concatenare il testo già letto con la nuova parola letta e con uno spazio, e assegnare il risultato della concatenazione alla variabile `testo`. Tale istruzione ha pertanto un ruolo analogo all'assegnazione:

```
somma = somma + x;
```

Anche in questo caso si ricorre all'impiego di un sottoprogramma non potendo usare una normale istruzione di assegnazione.

#### Esempio 8.4

Volendo verificare prima di effettuare la concatenazione:

```
strcat(testo, parola);
```

che vi sia sufficiente spazio libero nella variabile `testo`, si può utilizzare il sottoprogramma di tipo funzione `strlen` nel modo seguente:

```
if ( strlen(testo)+strlen(parola)+1 <= MAX_TESTO ) {
    strcat(testo, parola);
    strcat(testo, " ");
}
```

#### Esempio 8.5

Scrivere un programma che legge una parola, seguita da un intero  $N$ , seguito da un testo di  $N$  parole e che stampa il numero di occorrenze della prima parola letta nel testo.

Per chiarire il significato della specifica individuati i seguenti casi di test:

```
input:  prova 8 questa è una prova che riguarda le stringhe
output: 1
input:  testo 11 questo testo è un esempio di testo che contiene due occorrenze
output: 2
input:  xxx 6 questo è un testo di prova
output: 0
```

Il programma risolutivo richiede, dopo la lettura della prima parola e della lunghezza del testo, la scansione del testo e il conteggio di parole uguali alla prima parola letta. Si ottiene pertanto il programma:

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

main () {
    int n, i;
    int const MAX_PAROLA = 100;
    char prima_parola[MAX_PAROLA+1], parola_generica[MAX_PAROLA+1];
    int occorrenze;

    cin >> prima_parola,
    cin >> n;
    occorrenze = 0;
    for ( i=0; i<n; i++ ) {
        cin >> parola_generica;
        if ( !strcmp(prima_parola, parola_generica) ) // se sono uguali
            occorrenze++;
    }

    cout << "Il numero di occorrenze e': " << occorrenze << endl;
}
```

Per riconoscere gli schemi usati si confronti la soluzione con un algoritmo che conta il numero di occorrenze di un valore *k* in una lista di numeri:

```
cin >> k,
cin >> n;
occorrenze = 0;
for ( i=0; i<n; i++ ) {
    cin >> x;
    if ( x == k ) // se sono uguali
        occorrenze++;
}
```

In entrambi i casi si è usata la scansione di una lista con accumulazione su un contatore e si è proceduto a una fusione della fase di input e della fase di elaborazione della lista.

### 8.1.5 Output di stringhe

Per aggiungere allo standard output una stringa si può usare una normale istruzione di output. Ad esempio l'istruzione:

```
cout << testo;
```

aggiunge allo standard output la stringa contenuta nella variabile `testo`. Si noti che, come in tutti gli altri casi, affinché la stampa venga effettuata correttamente, la stringa memorizzata nella variabile `testo` deve essere terminata dal carattere nullo.

### 8.1.6 Ordinamento di stringhe

Un'operazione che è spesso richiesta quando si manipolano stringhe è quella di confrontare due stringhe per stabilire quale delle due sia minore rispetto all'ordinamento lessicografico (cioè quello che con termine comune è detto ordinamento alfabetico).

A tal fine non può essere usata la funzione di libreria `strcmp` nella forma descritta in precedenza che consente solo di stabilire se due stringhe sono uguali o diverse. Tuttavia, il prototipo mostrato in precedenza è una forma

semplificata del vero prototipo della funzione `strcmp`. Tale forma semplificata è efficace se la funzione deve essere usata per effettuare un test di uguaglianza, ma è necessario fare riferimento al prototipo effettivo di `strcmp` per poter usare tale funzione al fine di stabilire l'ordinamento lessicografico di due stringhe.

Prima di presentare il prototipo completo di `strcmp`, occorre ricordare che nel linguaggio C i valori numerici possono essere usati in sostituzione dei valori logici, con la convenzione che il valore numerico 0 si comporta come il valore logico **falso** e qualunque valore numerico diverso da 0 si comporta come il valore logico **vero**. In altre parole, l'espressione costante:

```
false || true
```

può essere scritta anche usando valori numerici:

```
0 || 1
```

oppure:

```
0 || -1
```

oppure usando qualsiasi valore diverso da 0 al posto del valore `true`.

Con questa premessa, il prototipo completo della funzione `strcmp` è il seguente:

```
int strcmp ( char s1[], char s2[] );
/* restituisce un valore negativo se s1 precede lessicograficamente s2,
   restituisce 0 se s1 è uguale a s2, restituisce un valore positivo se
   s1 segue lessicograficamente s2 */
```

Si noti come tale prototipo comprenda come caso particolare quello dato precedentemente. Infatti se le stringhe sono diverse (`s1` precede `s2` o `s1` segue `s2` nell'ordinamento lessicografico) viene restituito un valore numerico diverso da 0 che equivale al valore logico **vero**, se invece le stringhe sono uguali viene restituito il valore numerico 0 che equivale al valore logico **falso**.

### Esempio 8.6

Facendo riferimento al prototipo completo di `strcmp`, la funzione può essere usata per stabilire l'ordinamento lessicografico tra stringhe. Ad esempio, il seguente segmento di codice legge sullo standard input una lista di parole preceduta dalla sua lunghezza e stampa la parola che risulta minore rispetto all'ordinamento lessicografico. Si assuma che siano state dichiarate le variabili `parola`, `parola_minore`, `n`, `i`.

```
cin >> n;
if ( n > 0 ) {
    cin >> parola_minore;
    for ( i=1; i<n; i++ ) {
        cin >> parola;
        if ( strcmp(parola,parola_minore) < 0 )
            strcpy(parola_minore,parola);
    }
    cout << "La parola minore è :" << parola_minore << "\n";
}
else
    cout << "La lista e' vuota, non esiste una parola minore!\n";
```

## 8.2 Input/output con formato (a caratteri)

Quando sono state introdotte le istruzioni di input e di output, gli effetti della loro esecuzione sono stati spiegati descrivendo lo standard input e lo standard output come sequenze di valori. Tale spiegazione è sufficiente fin tanto che il formato dei dati di ingresso e di uscita non risulta troppo complesso. Vi sono tuttavia alcuni casi in cui l'effetto delle istruzioni di input e di output non si può spiegare descrivendo lo standard input e lo standard output come sequenze di valori, ma bisogna ricorrere a una descrizione che corrisponda meglio alla realtà.

### 8.2.1 Output

Cominciamo con il caso più semplice: lo standard output.

Se l'esecuzione di un programma produce il seguente output:

```
x y
0 0
1 2
2 5
3 10
4 17
```

è certamente possibile dire che esso aggiunge allo standard output la sequenza di 12 valori (tra parentesi il tipo del valore):

```
x (carattere)
y (carattere)
0 (numero)
0 (numero)
1 (numero)
2 (numero)
2 (numero)
5 (numero)
3 (numero)
10 (numero)
4 (numero)
17 (numero)
```

È tuttavia ovvio che descrivere lo standard output in questo modo non dice tutto, e di fatto la sequenza di istruzioni:

```
cout << 'x' ;
cout << 'y' ;
cout << 0 ;
cout << 0 ;
cout << 1 ;
cout << 2 ;
cout << 2 ;
cout << 5 ;
cout << 3 ;
cout << 10 ;
cout << 4 ;
cout << 17 ;
```

genera il seguente output:

```
xy001225310417
```

Per produrre infatti l'output su due colonne sopra riportato bisogna aggiungere, nei punti opportuni, dei caratteri newline, dei caratteri tab o dei caratteri bianchi.

Un ulteriore esempio, che mostra come sia insufficiente la descrizione dello standard output in termini di sequenza di valori, è il caso di stampa di un valore reale. Se lo standard output fosse una sequenza di valori, l'istruzione:

```
cout << 3.000018563 ;
```

dovrebbe generare l'output:

```
3.000018563
```

Invece, eseguendo la precedente istruzione di uscita, viene aggiunto allo standard output il numero:

Tipo dell'argomento di cout	Caratteri aggiunti allo standard output
carattere	lo stesso carattere
stringa	la stessa stringa
numeri interi	rappresentazione posizionale in base 10
numeri reali	rappresentazione posizionale in base 10 con punto decimale e numero prefissato di cifre frazionarie (tipicamente 5)

Tabella 8.1: Regole di conversione valori-caratteri impiegate nell'istruzione di output.

```
3.00002
```

I due esempi illustrati mostrano come, in casi non banali, sia necessario descrivere le istruzioni di output come istruzioni che aggiungono allo standard output caratteri e non valori. Di conseguenza lo standard output risulta essere in realtà una sequenza di caratteri e non di valori.

La distinzione può sembrare capziosa perchè noi siamo abituati a rappresentare i valori attraverso sequenze di caratteri (la parole sono sequenze di lettere, i numeri sono sequenza di cifre con l'aggiunta di caratteri come il segno e il punto decimale, ecc.) e dunque tendiamo a non dare importanza alle regole di conversione da valori a caratteri e viceversa che invece vanno prese in esplicita considerazione quando si tratta di programmare un esecutore automatico.

In effetti, la distinzione è sostanzialmente inutile in tutti i casi in cui le regole di conversione tra valori e caratteri sono semplici e intuitive, ed è per questo motivo che, limitandoci a tali casi semplici, abbiamo potuto introdurre le istruzioni di output in termini di sequenze di valori.

Volendo però dare ragione degli effetti delle istruzioni di output in casi più complessi come quelli riportati sopra, occorre fare riferimento in modo esplicito alle regole di conversione tra valori e caratteri utilizzate dall'istruzione cout.

Tali regole sono riportate nella tabella 8.1 per i tipi di dato che possono essere usati come argomento dell'istruzione di output.

## 8.2.2 Spiegazione degli esempi

È possibile ora dare ragione del perchè nel primo esempio riportato in precedenza l'output sia:

```
xy001225310417
```

Applicando infatti alla sequenza di istruzioni di output le regole riportate nella tabella 8.1, si verifica che vengono aggiunti allo standard output i caratteri 'x' e 'y' seguiti dalle cifre che rappresentano i numeri 0, 0, 1, 2, 2, 5, 3, 10, 4, 17. Poiché le istruzioni non lo prevedono non vengono inseriti spazi tra tali valori, e tantomeno vengono inseriti dei caratteri newline o dei tab per incolonnare i valori stessi.

Analogamente, nel secondo esempio il valore 3.000018563 viene convertito in rappresentazione posizionale con sole 5 cifre frazionarie e nella conversione viene operato un arrotondamento per eccesso sulla quinta cifra frazionaria.

## 8.2.3 Input

Anche per quanto riguarda l'input la descrizione in termini di sequenza di valori è adatta a descrivere solo situazioni relativamente semplici come ad esempio la lettura di una sequenza di numeri interi, o la lettura di una sequenza di parole.

Si consideri tuttavia il seguente segmento di codice:

```
const int MAX = 10;
char stringa[MAX];
for ( i=0; i<MAX; i++ )
    cin >> stringa[i];
for ( i=0; i<MAX; i++ )
    cout << stringa[i];
```

Tipo dell'argomento di cin	Caratteri usati per acquisire il valore
carattere	primo carattere diverso da uno spazio
stringa	prima stringa di caratteri consecutivi diversi dallo spazio
numeri interi	prima stringa di cifre consecutive, eventualmente precedute dal segno; se il primo carattere diverso dallo spazio non è una cifra il comportamento è indefinito
numeri reali	prima stringa di cifre consecutive eventualmente comprendenti il punto decimale, ed eventualmente precedute dal segno o dal punto decimale (viene in ogni caso presa in considerazione solo la prima occorrenza del punto decimale); se il primo carattere diverso dallo spazio non è una cifra il comportamento è indefinito

Tabella 8.2: Regole di conversione caratteri-valori impiegate nell'istruzione di input.

Se lo standard input contiene i caratteri:

```
a b c d \b \n e f g h \b \n i j k l \b \n
```

sullo standard output viene generata la sequenza di caratteri:

```
abcdefghijkl
```

contrariamente a quanto ci aspetteremmo, dal momento che sullo standard input erano presenti anche caratteri bianchi e *newline*.

Il motivo di questo comportamento è che anche gli effetti delle istruzioni di input devono essere descritti in termini di acquisizione di sequenze di caratteri (e non di valori) secondo determinate regole di conversione. Nel caso dell'input tali regole cercano di rendere il comportamento delle istruzioni quanto più simile possibile all'acquisizione di valori. Tuttavia vi sono casi in cui tale spiegazione è insufficiente e bisogna applicare esplicitamente le regole di conversione riportate nella tabella 8.2 per i tipi di dato che possono essere usati come argomento dell'istruzione di input.

È opportuno fare alcune osservazioni sul contenuto della tabella.

Prima di tutto occorre ricordare che le istruzioni di input consumano i caratteri seguendo l'ordine con cui essi sono presenti sullo standard input. Questo significa che quando nella tabella viene indicato che la conversione riguarda il primo carattere con certe caratteristiche, i caratteri eventualmente precedenti vengono saltati. Essi cioè vengono consumati senza produrre effetto.

La seconda osservazione riguarda la caratteristica comune a tutti i casi riportati in tabella: gli spazi presenti sullo standard input vengono sempre saltati. Occorre a questo proposito chiarire che per spazio non si intende solo il carattere spazio bianco, ma anche tutti quei caratteri che producono una separazione quando vengono aggiunti allo standard output. Oltre allo spazio bianco sono pertanto considerati spazi anche il carattere di *newline* e il carattere *tab*.

Il fatto che gli spazi vengano sempre saltati sembra rendere impossibile effettuare operazioni come: contare gli spazi bianchi presenti sullo standard input all'inizio di una riga, contare le righe di un testo fornito in input, ecc. In effetti, tali operazioni non possono essere fatti usando l'istruzione di input, a meno di non cambiare le regole di conversione che essa applica. Tale modifica è possibile e si rimanda a un manuale di C++ per ulteriori dettagli. Introduciamo comunque nel seguito un'istruzione che permette di leggere un'intera linea sullo standard input, inclusi eventuali spazi presenti. Usando tale istruzione è possibile effettuare qualunque operazione riguardi gli spazi presenti sullo standard input.

Venendo alle ultime due righe della tabella, occorre innanzitutto sottolineare come, quando si leggono dati numerici, la presenza sullo standard input di caratteri che non siano cifre, o segno, o, solo nel caso di numeri reali, punto decimale, vada assolutamente evitata perché può produrre il mal funzionamento del programma.

Infine, per chiarire ulteriormente le regole applicate nella lettura di dati numerici, è opportuno discutere alcuni esempi.

Consideriamo il segmento di programma:

```
int a;
```

```
double x;
char ch;
cin >> a;
cin >> x;
cin >> ch;
```

Se sullo standard input sono presenti i caratteri:

```
\b 1 0 . 0 3 . 0 3 \n
```

la prima istruzione di input salta lo spazio bianco iniziale e assegna alla variabile `a` il valore `10` consumando i primi tre caratteri sullo standard input; la seconda istruzione assegna alla variabile `x` il valore `0.03` consumando altri tre caratteri sullo standard input; la terza istruzione assegna alla variabile `ch` il valore `'.'` (carattere punto) e consuma un ulteriore carattere; successive istruzioni di input partono dal carattere `'0'` successivo al secondo punto.

Se sullo standard input sono presenti i caratteri:

```
5 0 8 \n - 9 8 \n . 0 3 \n
```

la prima istruzione di input assegna alla variabile `a` il valore `508` consumando i primi tre caratteri sullo standard input; la seconda istruzione salta il carattere newline e assegna alla variabile `x` il valore `-98`, consumando complessivamente altri quattro caratteri sullo standard input; la terza istruzione salta il secondo carattere newline e assegna alla variabile `ch` il valore `'.'` (carattere punto), consumando altri due ulteriori caratteri; successive istruzioni di input partono dal carattere `'0'` successivo al punto.

### 8.2.4 Altre istruzioni di input

Sono disponibili altre istruzioni di input che hanno una forma simile alla chiamata di sottoprogramma, con la differenza che il nome del sottoprogramma deve essere preceduto dal nome `cin` separato da un punto. Tali istruzioni, analgamente ai sottoprogrammi possono essere descritte fornendo un prototipo che indica se si tratta di funzioni o di procedure e se vi sono parametri.

#### Getline

L'istruzione `getline` viene descritta attraverso il seguente prototipo:

```
void getline ( char line[], int maxlen );
/* consuma tutti i caratteri presenti sullo standard input fino al primo
   carattere newline, memorizza i caratteri consumati (escluso il newline)
   nell'array line terminandoli con un carattere nullo; se la lunghezza
   della linea supera maxlen-1 caratteri, memorizza nell'array solo i primi
   maxlen-1 caratteri seguiti dal carattere nullo */
```

e viene indicata nel programma con la notazione:

```
cin.getline(nome_array, valore_intero);
```

dove `nome_array` e `valore_intero` sono i parametri effettivi della chiamata, rispettivamente di tipo array di caratteri e intero.

L'istruzione `getline` consente di leggere dallo standard input anche gli spazi bianchi e i tab eventualmente presenti su una linea. Essa non consente di leggere esplicitamente i newline, ma poiché ogni sua esecuzione consuma esattamente un newline dopo l'ultimo carattere letto, essa di fatto consente di tenere conto anche dei newline presenti sullo standard input.

#### Esempio 8.7

Il segmento di codice:

```

int n, i, j, n_bianchi;
const int MAX_LINEA = 250;
char linea[MAX_LINEA+1];
cin >> n;
n_bianchi = 0;
for ( i=0; i<n; i++ ) {
    cin.getline(linea,MAX_LINEA+1);
    j = 0;
    while ( linea[j] != '\0' ) {
        if ( linea[j]==' ' ) n_bianchi++;
        j++;
    }
}
cout << n_bianchi;

```

conta i caratteri bianchi presenti in un testo di n linee (il valore di n viene fornito sullo standard input prima del testo).

Se quindi lo standard input contiene i caratteri:

```
5 \b a a \n b b \b \n c \b c \n \b \b \b \n e e e \n
```

il programma, dopo aver letto il valore 5 nella variabile n, legge nell'array linea: la prima volta la stringa " aa", la seconda volta la stringa "bb ", la terza volta la stringa "c c", la quarta volta la stringa " " (3 spazi bianchi), la quinta volta la stringa "eee", e stampa il valore 6.

Se invece lo standard input contiene i caratteri:

```
3 \n \n c c c \n
```

il programma, dopo aver letto il valore 3 nella variabile n, legge nell'array linea: la prima volta la stringa "" (stringa vuota), la seconda volta ancora la stringa "", la terza volta la stringa "ccc", e stampa il valore 0.

## Eof

La lettura di dati terminati da un'informazione tappo ha il problema che non sempre è facile determinare quale valore usare come tappo perchè tale valore va escluso a priori tra quelli di interesse per il programma. Il problema è particolarmente grave nel caso di lettura di testi perchè da un lato in questi casi non è ragionevole assumere di conoscere la lunghezza del testo prima di elaborarlo (si pensi ad esempio di dover fornire il numero di parole presenti in una relazione di 10 pagine!), e dall'altro non è facile trovare un carattere che non venga mai usato in nessun testo per assegnargli il ruolo di informazione tappo.

Per questo motivo tutti i linguaggi introducono la possibilità di inserire sullo standard input un carattere speciale che ha il compito di segnalare la fine dell'input stesso. Per vari motivi il programma non può controllare la presenza di tale carattere con un test esplicito, ma deve ricorrere a un'istruzione descritta dal seguente prototipo:

```

bool eof ( );
/* restituisce true se nell'ultima operazione di lettura è stato consumato
   il carattere di fine file, restituisce false altrimenti */

```

Come nel caso di getline, tale istruzione viene invocata mediante una chiamata preceduta da cin separato da un punto. Questa volta, tuttavia, il prototipo indica che si tratta di una funzione e la chiamata non è pertanto un'istruzione autonoma, ma va inserita all'interno di un'espressione.

## Esempio 8.8

Si supponga di voler contare il numero di spazi in un testo (vedi esempio precedente), ma che non si disponga del numero di righe da cui il testo è costituito. Se dopo aver digitato il testo si aggiunge dopo l'ultimo newline il carattere speciale di fine file (che si ottiene su quasi tutti i sistemi premendo insieme i tasti `Ctrl` e `z`), si può usare l'istruzione eof per far terminare il ciclo che legge il file, ottenendo il segmento di codice:

```

int i, j, n_bianchi;
const int MAX_LINEA = 250;
char linea[MAX_LINEA+1];
n_bianchi = 0;
cin.getline(linea,MAX_LINEA+1);
while ( !cin.eof() ) {
    j = 0;
    while ( linea[j] != '\0' ) {
        if ( linea[j]==' ' ) n_bianchi++;
        j++;
    }
    cin.getline(linea,MAX_LINEA+1);
}
cout << n_bianchi;

```

Si osservi che il segmento segue lo schema della lettura di una lista terminata da un'informazione tappo che riportiamo di seguito per comodità di lettura:

```

acquisisci in a[0] il primo valore;
i = 0;
while ( a[i] != TAPPO ) {
    /* a[i] contiene l'elemento di posto i+1 */
    i++;
    acquisisci in a[i] il valore successivo ;
}
/* i contiene la lunghezza della lista */

```

Si noti che nel caso in esame la lista da leggere è la lista delle righe del testo e che manca l'indice *i* perché le righe vengono immediatamente elaborate, e l'array *linea* usato per leggerle viene riutilizzato ad ogni lettura. Si tratta in effetti di un esempio di fusione tra input ed elaborazione secondo uno schema del tipo:

```

acquisisci in il primo valore;
while ( input non terminato ) {
    elabora valore acquisito;
    acquisisci il valore successivo;
}

```

Si noti infine come, in accordo alle proprietà di lettura con informazione tappo, è necessario effettuare una lettura prima di cominciare il ciclo **while** per verificare se il testo non sia vuoto (lo standard input potrebbe contenere subito il carattere di fine file). Inoltre, la lettura interna al ciclo è posta in fondo al suo corpo, in modo da controllare subito dopo di non aver incontrato il fine file (condizione di uscita del ciclo).

### 8.2.5 Input/output da file

Fino ad ora abbiamo visto come acquisire dati dallo standard input e aggiungere dati sullo standard output. Vi sono tuttavia casi in cui è più conveniente acquisire dati da un file o aggiungere dati a un file. Si pensi ad esempio al caso in cui, in input o in output, occorre gestire grandi quantità di dati che non possono essere gestiti manualmente, o quando i dati vanno conservati per un uso successivo.

I moderni linguaggi di programmazione risolvono il problema mediante meccanismi del tutto simili a quelli usati per gestire lo standard input e lo standard output in modo che i programmi risultino largamente indipendenti dall'origine e dalla destinazione effettiva dei dati.

Questo risultato si è ottenuto osservando che lo standard input e lo standard output sono sequenze di caratteri e che pertanto risultano equivalenti a un file che contenga esclusivamente caratteri. Un tale file, denominato *text file*, può anche essere generato o letto dagli utenti mediante programmi già pronti come ad esempio il programma *notepad* nel caso di windows o il programma *vi* nel caso di UNIX o LINUX.

Sulla base di questa equivalenza, sono stati predefiniti due tipi di dato, denominati `ifstream` e `ofstream`, che permettono di dichiarare variabili che possono essere associate a file, usate allo stesso modo di `cin` e `cout`, rispettivamente. In altre parole sulle variabili `ifstream` si possono usare le operazioni:

- `>>` e `getline` per acquisire dati da un file, con le stesse regole viste in precedenza, caratteri, stringhe e numeri;
- l'operazione `eof` per individuare la fine del file.

Viceversa, sulle variabili `ofstream` si può usare l'operazione `<<` per aggungere caratteri ad un file, sempre con le stesse regole già viste.

Per usare i tipi `ifstream` e `ofstream` occorre includere nel programma il file header `fstream.h`. Inoltre prima di effettuare qualunque operazione sulle variabili di entrambi i tipi occorre associarle ad un file mediante l'operazione `open` il cui prototipo è:

```
void open ( char nome_file[] );
/* associa nome_file (stringa di caratteri contenente il nome
secondo le convenzioni del file system) alla variabile ifstream
o ofstream a cui e' applicata l'operazione; l'associazione puo'
terminare con successo o con fallimento (vedi operazione
is_open);
nome_file può comprendere anche il path relativo o assoluto del
file; in caso di path relativo la directory di default viene
decisa secondo regole che dipendono dall'implementazione del
linguaggio usata */
```

È anche disponibile un'istruzione complementare ad `open` che annulla l'associazione di una variabile di tipo `ifstream` o `ofstream` con un file. L'istruzione ha il prototipo:

```
void close ( );
/* annulla l'associazione con un file della variabile di tipo ifstream
o ofstream su cui viene applicata */
```

Anche se la terminazione di un programma annulla ogni associazione a file di variabili di tipo `ifstream` o `ofstream`, è buona norma di programmazione annullare esplicitamente l'associazione tra una variabile e un file con l'istruzione `close` quando tale associazione non è più utile.

### Esempio 8.9

Il segmento di codice:

```
int n;
int x;
int somma;
cin >> n;
somma = 0;
for ( i=0; i<n; i++ ) {
    cin >> x;
    somma = somma + x;
}
cout << somma;
```

legge dallo standard input una lista di numeri preceduta dalla sua lunghezza, ne calcola la somma e aggiunge tale somma allo standard output. Volendo effettuare la stessa operazione su una lista di memorizzata sul file `c:\my_data\lista.txt`, il segmento di codice varrebbe così modificato:

```

ifstream in_file;
int n;
int x;
int somma;
in_file.open("c:\\my_data\\lista.txt");
in_file >> n;
somma = 0;
for ( i=0; i<n; i++ ) {
    in_file >> x;
    somma = somma + x;
}
cout << somma;
in_file.close();

```

Si noti che, ovviamente, all'inizio del file che contiene il codice va inserita la direttiva:

```
# include <fstream.h>
```

Inoltre si assume che nella directory `c:\\my_data` esista un text file denominato `lista.txt` e che tale file contiene una lista di numeri preceduta dalla sua lunghezza. In altre parole, il contenuto del file `lista.txt` deve coincidere con quello che l'utente avrebbe dovuto digitare sul dispositivo associato allo standard input nel caso del primo segmento di codice.

Si noti infine che le istruzioni `open` e `close` hanno la forma di chiamate a procedura, con la differenza che devono essere precedute dal nome della variabile di tipo `ifstream` su cui vengono applicate, separata da un punto. Si tratta di una situazione del tutto analoga a quelle viste in precedenza per le operazioni `getline` e `eof` applicate su `cin`.

### Esempio 8.10

Nel caso di variabili `ofstream`, l'istruzione di associazione al file ha una forma del tutto simile al caso `ifstream`. Il seguente segmento di codice genera il text file `c:\\my_data\\tabella.txt` contenente la tabella della moltiplicazione tra cifre decimali.

```

ofstream out_file;
int i, j;
out_file.open("c:\\my_data\\tabella.txt");
for ( i=0; i<10; i++) {
    for ( j=0; j<10; j++ ) {
        out_file << "\\t";
        out_file << i*j;
    }
    out_file << "\\n";
}
out_file.close();

```

## 8.2.6 Gestione degli errori

L'esecuzione dell'istruzione `open` su una variabile di tipo `ifstream` richiede che, nella directory indicata (esplicitamente se nell'istruzione è incluso il path assoluto, implicitamente se è indicato un path relativo o non è indicato alcun path), il file da associare esista e che l'utente abbia il diritto di leggerne il contenuto. È ovvio infatti che il file deve già esistere per poter essere letto.

Se tale condizione non si verifica, l'istruzione `open` non produce alcuna associazione della variabile `ifstream` e le successive operazioni su di essa non sono lecite. Poiché l'eventualità che l'associazione non vada a buon fine non è rara (basta ad esempio che il file o una directory del path sia stata rinominata dopo che il programma è stato compilato) è opportuno che il programma controlli la buona riuscita dell'istruzione `open` prima di proseguire.

A tal fine è disponibile l'istruzione `is_open` caratterizzata dal seguente prototipo:

```

bool is_open ( );
/* restituisce true se la variabile di tipo ifstream o ofstream su cui
   e' è validamente associata ad un file, restituisce false altrimenti */

```

### Esempio 8.11

Quando si effettua l'associazione di una variabile di tipo `ifstream` o è opportuno farlo usando un segmento di codice simile al seguente:

```

ifstream in_file;
in_file.open("c:\my_data\lista.txt");
if ( !in_file.is_open() ) {
    cout << "errore --- apertura del file fallita\n";
    exit(1);
}

```

Si noti che la verifica di corretta associazione delle variabili di tipo `ofstream` si effettua in maniera del tutto analoga. In questo caso, tuttavia, l'eventualità di errore è molto minore perché l'associazione di un file a una variabile `ofstream` produce la creazione del file se il file non esiste e la sua cancellazione e riscrittura se esiste. La possibilità di errore è quindi legata solo all'impossibilità di creare il file che è un evento raro (accade quando il disco è pieno o quando l'utente non ha i diritti necessari a scrivere sul disco).

### Esempio 8.12

Forniamo ancora un esempio completo di utilizzo delle variabili `ifstream` e `ofstream`.

Il seguente programma legge un testo e lo ristampa su righe con lunghezza massima assegnata. Il testo viene letto da un file e viene ristampato su un altro file. I nomi dei due file vengono forniti dall'utente sullo standard input. Inoltre eventuali errori vengono segnalati da messaggi stampati sullo standard output.

```

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>

int main()
{
// ----- DICHIARAZIONI -----

    const int MAX_PAROLA = 100;
    char parola[MAX_PAROLA];

    const int MAX_NOME_FILE = 35;
    char nome_in_file[MAX_NOME_FILE], nome_out_file[MAX_NOME_FILE];

    int linea_len, max_len;

    ifstream in_file;
    ofstream out_file;

// ----- INIZIALIZZAZIONI -----

    cout << "Dammi i nomi dei file di input e di output: ";
    cin >> nome_in_file;
    cin >> nome_out_file;

```

```

in_file.open(nome_in_file);
if ( !in_file.is_open() ) {
    cout << "errore --- non posso aprire il file di input\n";
    system("PAUSE");
    exit(1);
}
out_file.open(nome_out_file);
if ( !out_file.is_open() ) {
    cout << "errore --- non posso aprire il file di output\n";
    system("PAUSE");
    exit(1);
}

cout << "Dammi la lunghezza massima delle linee del file di output: ";
cin >> max_len;

// ----- ALGORITMO -----

linea_len = 0;
in_file >> parola;
while ( !in_file.eof() ) {
    in_file >> parola;
    if ( linea_len + strlen(parola) > max_len ) {
        out_file << "\n";
        linea_len = 0;
    }
    out_file << parola << " ";
    linea_len = linea_len + strlen(parola);
}
out_file << "\n";

// ----- CHIUSURA -----

in_file.close();
out_file.close();

return 0;
}

```

Vale la pena fare le seguenti osservazioni:

- l'algoritmo è una fusione di input, elaborazione e output e segue lo schema dell'elaborazione di una lista (di parole) con informazione tappo (il carattere di fine file); l'elaborazione consiste essenzialmente nel ricopiare le parole lette in uscita separandole con uno spazio;
- la variabile `linea_len` è usata come accumulatore, con la variante che viene riazzerata ogni volta che viene superata la lunghezza massima ammessa per una linea del testo in uscita;
- i nomi dei file di ingresso e di uscita sono forniti dall'utente e letti in due variabili stringa; le due variabili stringa sono usate nelle istruzioni di `open` per associare le variabili `in_file` e `out_file` ai corrispondenti file;
- grazie all'impiego di variabili `ifstream` e `ofstream` il programma acquisisce dati dallo standard input (i nomi dei file, la lunghezza massima) e da un file (il testo di ingresso) e genera dati sullo standard output (i messaggi di errore) e su un file (il testo di uscita).

### 8.3 Array multidimensionali

Nel linguaggio C è possibile dichiarare e manipolare variabili array con più di una dimensione. Si parla in tal caso di array multidimensionali.

La dichiarazione di un array multidimensionale è del tutto simile a quella degli array ad una dimensione (detti pertanto monodimensionali), con l'estensione relativa a ciascuna dimensione racchiusa tra parentesi quadre.

Ad esempio le dichiarazioni:

```
const int MAX_DIM1 = 20;
const int MAX_DIM2 = 10;
const int MAX_DIM3 = 10;
const int MAX_DIM4 = 4;
char     matrice[MAX_DIM2][MAX_DIM1];
int      cubo[MAX_DIM3][MAX_DIM2][MAX_DIM1];
double   ipercubo[MAX_DIM4][MAX_DIM3][MAX_DIM2][MAX_DIM1];
```

dichiarano, rispettivamente, un array bidimensionale (normalmente indicato con il termine *matrice*) di  $10 \times 20 = 200$  caratteri, un array tridimensionale di  $10 \times 10 \times 20 = 2000$  interi, un array a quattro dimensioni di  $4 \times 10 \times 10 \times 20 = 8000$  reali.

Nelle istruzioni le celle di un array multidimensionale si indicano elencando dopo il nome dell'array tutti gli indici, ciascuno racchiuso tra parentesi quadre. Come per gli array monodimensionali, gli indici sono espressioni semplici o composte, costanti o non costanti.

Ad esempio, assumendo che *i*, *j*, *k* sono variabili intere opportunamente definite, le istruzioni:

```
cout << matrice[i+1][j];
cubo[0][0][k] = cubo[1][0][k] + cubo[2][0][k];
ipercubo[i+j][i-j][k][k] = 0.001;
```

rispettivamente:

- aggiungono allo standard output il valore memorizzato nella cella dell'array *mat* appartenente alla riga di indice *i+1* e alla colonna *j*;
- assegnano alla cella dell'array *cubo* di indici 0, 0, *k*, la somma dei valori contenuti nelle celle dello stesso array di indici 1, 0, *k*, e 2, 0, *k*;
- assegnano alla cella dell'array *ipercubo* di indici *i+j*, *i-j*, *k*, *k* il valore costante 0.001.

### 8.4 Parametri di scambio array multidimensionali

Come abbiamo visto un array monodimensionale è caratterizzato da un tipo base (il tipo dei valori contenuti nelle singole celle) e da un'estensione (il numero di celle da cui è costituito l'array).

Secondo le regole del linguaggio, anche gli array a più dimensioni sono considerati array monodimensionali, ma con la proprietà che il loro tipo base è a sua volta un array, eventualmente anche esso a più dimensioni. In altri termini, una matrice viene trattata come un array il cui tipo base è un array monodimensionale, un array tridimensionale viene trattato come un array il cui tipo base è una matrice, e così via.

Questa caratteristica è ricca di conseguenze per quanto riguarda il passaggio di array multidimensionali ai sottoprogrammi.

Consideriamo innanzi tutto la dichiarazione (nel seguito useremo sempre array le cui celle elementari sono interi, ma il discorso è indipendente da questo):

```
int a1[MAX_DIM1];
```

Il tipo base di *a1* è **int** e la sua estensione è il valore di *MAX\_DIM1*. Consideriamo quindi la dichiarazione:

```
int a2[MAX_DIM2][MAX_DIM1];
```

Questa volta il tipo base è `int [MAX_DIM1]` (volendo significare con questa notazione che il tipo base è un array di `MAX_DIM1` interi) e l'estensione è il valore di `MAX_DIM2`. Consideriamo infine la dichiarazione:

```
int a3 [MAX_DIM3] [MAX_DIM2] [MAX_DIM1];
```

In questo caso il tipo base è `int [MAX_DIM2] [MAX_DIM1]` (volendo significare con questa notazione che il tipo base è una matrice di `MAX_DIM2 × MAX_DIM1` interi) e l'estensione è il valore di `MAX_DIM3`.

Da questi esempi risulta che il tipo base di un array è caratterizzato dal tipo delle celle elementari specificato all'inizio della dichiarazione e dalle estensioni, specificate in fondo alla dichiarazione, esclusa la prima che indica il numero di elementi dell'array (inteso come array monodimensionale di qualcosa).

Supponiamo ora, per fissare le idee, di avere una procedura di nome `proc1` con un paramtro array monodimensionale di nome `par1`. Il prototipo di tale procedura è (assumiamo sempre senza perdita di generalità che le celle elementari siano degli interi):

```
void proc1 ( int par[] );
```

Tenendo presente le osservazioni fatte sul tipo base di un array multidimensionale, si può comprendere come il prototipo di una procedura `proc2` con un parametro array bidimensionale `par2` va dichiarato nel modo seguente:

```
void proc2 ( int par2 [] [MAX_DIM1] );
```

dove `MAX_DIM1` è il numero di elementi del tipo base dell'array (che è per l'appunto un array monodimensionale di interi).

Analogamente, il prototipo di una procedura `proc3` con un parametro array tridimensionale `par3` va dichiarato nel modo seguente:

```
void proc3 ( int par3 [] [MAX_DIM2] [MAX_DIM1] );
```

dove `MAX_DIM2 × MAX_DIM1` è il numero di elementi del tipo base dell'array (che è per l'appunto un array bidimensionale di interi).

Nella chiamata a sottoprogramma, il parametro effettivo corrispondente a un array multidimensionale deve essere un array con pari numero di dimensioni. Le estensioni devono essere coincidenti con quelle dichiarate nel prototipo del sottoprogramma, tranne la prima estensione che può essere qualsiasi.

Si vede facilmente come il caso di array monodimensionali sia un caso particolare di quello descritto, dove il tipo base dell'array non è a sua volta un array e l'estensione può essere qualsiasi.

Si noti infine che le regole descritte hanno un interessante conseguenza. Si consideri l'array bidimensionale *matrice* dichiarato in precedenza. Indicando con `i1`, `i2` due espressioni qualsiasi di tipo intero di valore compatibile con le estensioni dichiarate per le due dimensioni di matrice, la notazione:

```
matrice[i2][i1]
```

denota, come già sappiamo, la casella dell'array bidimensionale associata agli indici `i1`, `i2`. D'altra parte, per quanto detto in precedenza, l'array *matrice* può essere visto come un array monodimensionale di array monodimensionali, a dunque la notazione:

```
matrice[i2]
```

denota l'elemento di indice `i2` di tale array monodimensionale che è appunto un array monodimensionale di `MAX_DIM1` elementi. Più esattamente, in coerenza con le regole già viste sui riferimenti agli array, tale notazione denota il riferimento a un array monodimensionale di `MAX_DIM1` elementi. Si noti che questo è perfettamente coerente con il significato della notazione a due indici, se si immagina di applicare gli indici nell'ordine da sinistra verso destra. Se infatti la notazione `matrice[i2]` equivale a riferire il nome di un array monodimensionale, la notazione `matrice[i2][i1]` rappresenta la cella di indice `i1` di tale array, che è appunto la cella di indice `i1` appartenente alla riga della matrice di indice `i2`.

Si noti anche che, per quanto detto, assumendo che la procedura `leggi_array_mono` abbia il prototipo:

```
void leggi_array_mono ( char stringa[] );
/* legge dallo standard input una stringa di caratteri e la memorizza
nell'array sringa terminandola con un carattere nullo */
```

il segmento di codice (la variabile `matrice` è quella dichiarata precedentemente):

```
for ( i=0; i<MAX_DIM2; i++ )  
    leggi_array_mono(matrice[i]);
```

è perfettamente legittimo e significa che il sottoprogramma `leggi_array_mono` riceve come parametro effettivo di volta in volta le singole righe della matrice, trattate come array monodimensionali di caratteri.