Capitolo 67

Gestione dei flussi di file in C

67.1 Concetti generali

«

Il linguaggio C ha un proprio modo per gestire i file che, per poter essere compreso, richiede l'introduzione di alcuni concetti, presentati in questo capitolo. Va osservato che lo standard del linguaggio C prevede i flussi di file, i quali però, in un sistema che si rifà al modello di Unix sono gestiti attraverso i descrittori di file. Per scrivere codice C che sia compatibile nel modo migliore con qualunque sistema operativo, occorre avvalersi soltanto dei flussi, a cui qui ci si riferisce.

67.1.1 Dal file al flusso di file

Dal punto di vista del programma scritto in linguaggio C, il file viene utilizzato in qualità di ***flusso logico di dati*** (*stream*), ovvero flusso di file. Per la precisione, un file viene aperto attribuendogli un puntatore che rappresenta il flusso di file relativo; quando poi il flusso viene chiuso, l'associazione con il file si conclude.

La gestione del flusso di file avviene in modo trasparente, con l'ausilio di funzioni standard, ma ciò implica la presenza di una sorta di tabellina contenente una serie di informazioni legate all'accesso al file. Questa tabellina è formata in modo differente, a seconda del contesto in cui ci si trova a compilare il programma, ma in generale dovrebbe contenere almeno alcune informazioni basilari: il riferimento a un array di caratteri usato in qualità di memoria tampone, assieme ai vari puntatori necessari alla sua gestione; il tipo di accesso al file; i riferimenti per accedere al file secondo le caratteristiche del sistema operativo.

Quella tabellina che raccoglie tutte le informazioni su un certo flusso di file è definita da una variabile strutturata, dalla quale deriva un tipo di dati dichiarato nel file di intestazione '`stdio.h`'. Il tipo di dati in questione è denominato '**`FILE`**'.

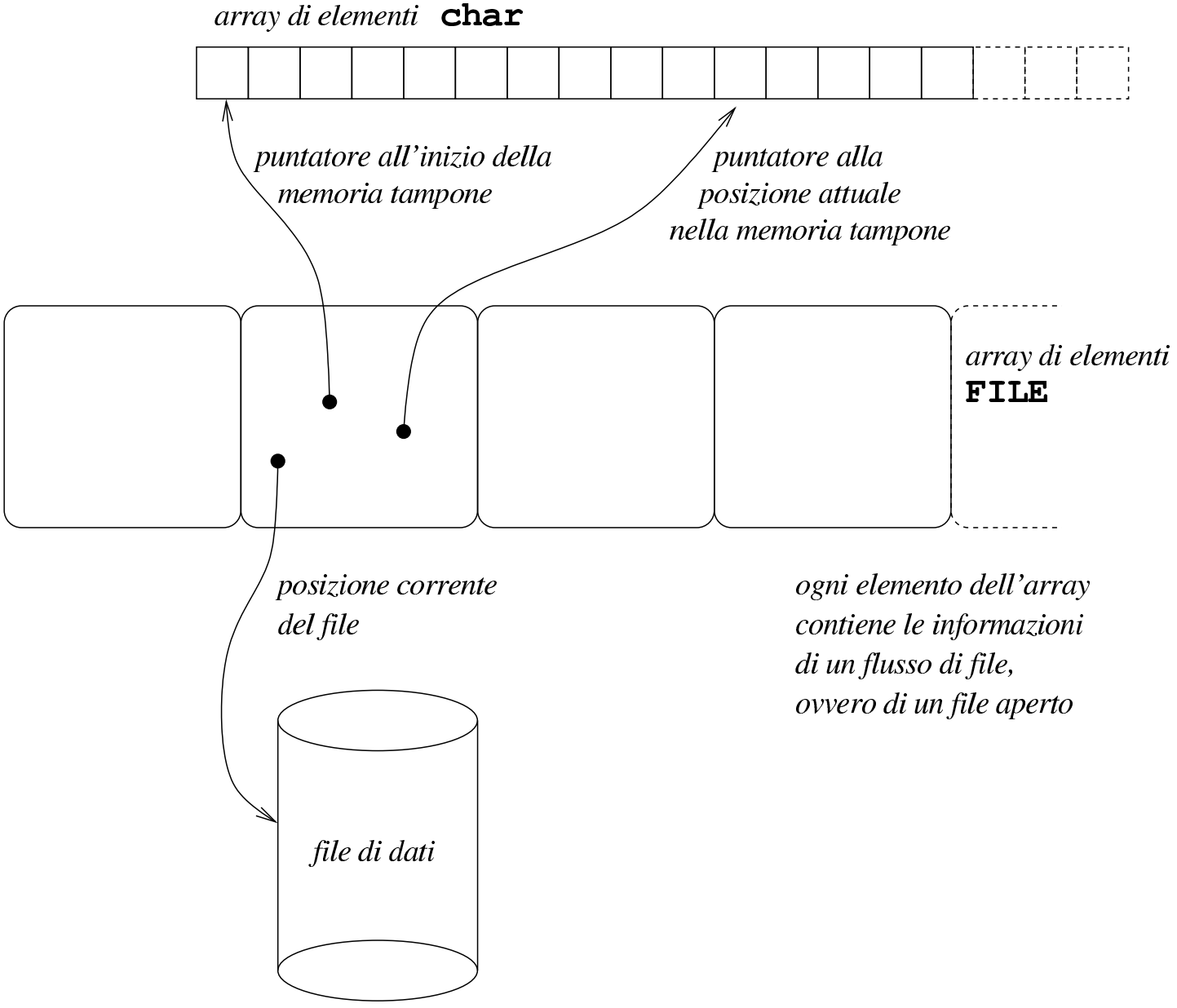
L'apertura di un file, attraverso le funzioni standard, coincide con l'ottenimento di un puntatore al tipo '**`FILE`**'; pertanto, questo puntatore rappresenta il flusso di file e tutti i riferimenti a tale flusso si fanno con quel puntatore.

La modalità di accesso al file distingue tra lettura, scrittura e scrittura in aggiunta, utilizzando una simbologia particolare per esprimerla. Lo specchietto successivo sintetizza le operazioni consentite in base

alla modalità utilizzata:[1]

`r`	`w`	`a`	`r+`	`w+`	`a+`	Annotazioni
X			X			Per aprire un file in lettura, con queste modalità, è necessario che esista già.
	X			X		Quando si apre un file in scrittura, con queste modalità, se il file non esiste viene creato al volo, se invece esiste già, il suo contenuto precedente viene eliminato.
X			X	X	X	Con queste modalità è possibile leggere il contenuto del file.
	X	X	X	X	X	Con queste modalità è possibile modificare il contenuto del file.
		X			X	Con queste modalità è possibile scrivere nel file soltanto aggiungendo dati in coda.

Figura 67.2. Rappresentazione intuitiva dell'associazione tra una variabile strutturata di tipo '**FILE**' e il file a cui fa riferimento. Qui viene ipotizzato un array di elementi di tipo '**FILE**', ma non è detto che l'organizzazione della libreria standard che si utilizza sia conforme a questa organizzazione.



67.1.2 File di testo e file binari

«

Il linguaggio C nasce per il sistema Unix, dove il file di testo ha una conformazione particolare che non è condivisa universalmente. Il file di testo in un sistema Unix o derivato è composto da una sequenza di caratteri (tradotti in byte),[2] dove la separazione tra le righe è segnalata dal codice *new-line*, corrispondente a <*LF*>, ovvero la sequenza '**\n**'.

Nei sistemi Dos e MS-Windows si ha una rappresentazione simile, dove però il codice di interruzione di riga è rappresentato dalla sequenza <*CR*><*LF*>. In altri sistemi si usano codice di interruzione di riga differenti e sono ammissibili forme molto diverse per rappresentare un file di testo.

Per questa ragione, il linguaggio C distingue l'accesso ai file attraverso due tipologie fondamentali: file di testo e file binari. In questo modo, quando si prevede un accesso in modalità testuale, la lettura e la scrittura del file avvengono attraverso una mediazione, tale da consentire al programmatore di trattare il file come se avesse la stessa rappresentazione di un sistema Unix. Naturalmente, in un sistema Unix e in qualunque altro sistema equivalente e conforme alla tradizione, non c'è distinzione tra l'accesso testuale ai file e quello binario.

Da quanto esposto vanno considerate due cose: quando si interviene su un file di testo, il codice corrispondente alla sequenza '**\n**' va inteso genericamente come codice di interruzione di riga; inoltre, il modo in cui si tiene traccia della posizione corrente all'interno di un file di testo non è predeterminabile, soprattutto perché non si può sapere quanti byte separano la fine di una riga dall'inizio della

successiva.

Il testo seguente è citato dalla documentazione standard *ISO/IEC 9899:TC2* e può servire per comprendere meglio il significato attribuito ai concetti di file di testo e di file binario:

> *A text stream is an ordered sequence of characters composed into lines, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is implementation-defined. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text in the host environment. Thus, there need not be a one- to-one correspondence between the characters in a stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consist only of printing characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is implementation-defined.*
>
> *A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream, under the same implementation. Such a stream may, however, have an implementation-defined number of null characters appended to the end of the stream.*

67.1.3 Fine del file

«

Nei documenti che trattano del linguaggio C si fa spesso riferimento alla macro-variabile ***EOF*** (dichiarata nel file '`stdio.h`'), in qualità di valore che si ottiene quando si tenta di leggere oltre la fine del file. La macro-variabile ***EOF*** corrisponde a un valore negativo che solitamente è −1, trattato come intero normale. Generalmente si può ottenere un valore di questo genere quando la lettura avviene carattere per carattere (inteso nel senso del tipo '**`char`**', corrispondente al byte), perché in questi casi il carattere letto viene convertito in un valore senza segno, esteso alla dimensione di un intero normale. In questo modo, nessun carattere potrebbe confondersi con un valore negativo di un intero di tipo '**`int`**'.

Quando però la lettura di un file avviene attraverso funzioni che leggono un carattere esteso alla volta (l'equivalente di un carattere '**`wchar_t`**'), queste restituiscono un valore di tipo differente ('**`wint_t`**') con cui si può rappresentare sia un carattere esteso, sia il valore rappresentato dalla macro-variabile ***WEOF*** che non individua alcun carattere esteso e rappresenta il raggiungimento della fine del file. A differenza di ***EOF***, il valore di ***WEOF*** potrebbe essere positivo o negativo indifferentemente, perché conta solo che si tratti di un valore che non corrisponde ad alcun carattere esteso.

Di norma, il raggiungimento della fine di un file viene annotato all'interno della variabile strutturata che controlla il flusso (a cui ci si riferisce con un puntatore di tipo '**`FILE *`**') e può essere interrogata con una funzione apposita. Naturalmente, l'uso di una funzione che porti alla modifica della posizione corrente, va ad azzerare tale indicazione.

67.1.4 Memoria tampone

I flussi di file possono disporre di una memoria tampone (*buffer*) che di norma è costituita da un array di caratteri ed è gestita da puntatori annotati all'interno delle variabili strutturate di tipo '**`FILE`**' associate ai flussi stessi.

Il programmatore ha la possibilità di controllare l'uso della memoria tampone, definendone la dimensione o arrivando a escluderla del tutto. In particolare, se si utilizza la memoria tampone, si può distinguere tra una gestione completa e una gestione a righe di testo.

L'uso della memoria tampone implica che le operazioni di scrittura possono avvenire con un certo ritardo. In generale, alla chiusura di un flusso di file si ottiene anche lo scarico della memoria tampone per ciò che riguarda le operazioni di scrittura ancora sospese; eventualmente è disponibile anche una funzione per richiedere espressamente l'esecuzione della scrittura in qualunque altro momento.

Va osservato che gli accessi ai file si prevedono in modo esclusivo; pertanto la gestione della memoria tampone è interna al programma. Per un accesso condiviso ai file la memoria tampone non può essere usata e comunque occorrono delle accortezze che le funzioni standard non possono offrire.

67.1.5 Flussi standard

«

Il linguaggio C prevede che ogni programma disponga, in modo predefinito, di tre flussi di file già costituiti: standard input, standard output e standard error. Il primo è predisposto per la lettura e di norma è collegato alla tastiera; il secondo e il terzo consentono solo la scrittura e sono collegati normalmente allo schermo.

Il fatto di disporre di tre flussi già in essere implica che ci siano tre puntatori di tipo '`FILE *`' già predisposti e associati correttamente alle strutture rispettive, per il controllo dei flussi di competenza. Va osservato che mentre i flussi standard non possono essere costituiti esplicitamente, potrebbero invece essere chiusi, oppure potrebbero essere riassegnati associandoli a file (o dispositivi) differenti.

L'associazione iniziale dei flussi standard a file o dispositivi dipende da ciò che succede in fase di avvio del programma (una shell potrebbe ridirigere i flussi a file diversi da quelli consueti). In condizioni normali, lo standard error è privo di memoria tampone, perché ciò che viene segnalato attraverso questo canale deve essere recepito il più presto possibile; per quanto riguarda invece gli altri due flussi, se questi non sono associati a dispositivi interattivi, di norma sono provvisti di memoria_tampone.

Rimane da chiarire in che modo il file corrispondente al flusso sia aperto: l'associazione a una modalità di accesso binaria o testuale dovrebbe dipendere dal contesto e precisamente da ciò che determina il sistema operativo. È comunque possibile cambiare espressamente tale modalità, nel caso ciò fosse auspicabile.

67.1.6 Orientamento

I dati scritti e letti da un file vengono gestiti sempre attraverso sequenze di byte. Quando si devono rappresentare «caratteri estesi», tali da non poter essere espressi in un solo byte, si usano delle sequenze multibyte, secondo una codifica che normalmente dipende dalla configurazione locale.

La codifica multibyte utilizzata può essere priva di stato, in quanto ogni carattere esteso ha la propria sequenza indipendente, oppure può richiedere, di volta in volta, la selezione di un sottoinsieme di caratteri differente (attraverso quello che viene chiamato *shift state*). In ogni caso, sia la scrittura, sia la lettura, richiede di tenere traccia dello stato di completamento e, se necessario, della modalità di interpretazione in corso (*shift state*). Queste informazioni possono essere raccolte in un'area di memoria organizzata secondo il tipo '**`mbstate_t`**' (*Multibyte state*) che di solito è strutturata in più componenti.

Nella variabile strutturata di tipo '**`FILE`**' che rappresenta un flusso aperto, usata per gestire l'accesso al file relativo, deve essere presente un componente di tipo '**`mbstate_t`**' per poter seguire lo stato di interpretazione di una sequenza multibyte.

Onde evitare confusione, un flusso di file (aperto in modo binario o testuale, indifferentemente), deve essere ***orientato***, nel senso che occorre stabilire se vada gestito a caratteri normali o estesi. In mancanza di una dichiarazione esplicita, l'orientamento viene definito in base all'uso del flusso attraverso funzioni specializzate per il trattamento di stringhe normali o di stringhe estese. Per esempio, si ottiene un orientamento orientato al byte (*byte-oriented*) se si utilizza la

funzione ***fprintf()*** (*file print formatted*), mentre si ottiene un orientamento esteso (*wide-oriented*) se si usa la funzione ***fwprintf()*** (*file wide print formatted*).

Una volta impostato l'orientamento, anche solo attraverso l'uso iniziale di una funzione invece di un'altra, questo può essere cambiato solo in modo esplicito, eventualmente riaprendo il flusso. Ma se questo cambiamento esplicito non viene eseguito, non è possibile utilizzare il flusso attraverso funzioni che non siano conformi all'orientamento esistente.

Si osservi che anche i tre flussi standard, all'inizio dell'esecuzione del programma, sono ancora privi di orientamento.

67.2 Utilizzo comune dei file

«

Nel linguaggio C, i file aperti sono flussi di file e l'apertura coincide con la predisposizione automatica di una variabile strutturata di tipo '**FILE**', a cui, di conseguenza, si fa riferimento attraverso un puntatore (di tipo '**FILE ***'). Di solito, questo puntatore viene chiamato discorsivamente «puntatore al file», ovvero *file pointer*.

Quando si vuole accedere a un file, così come per poter usare le funzioni che consentono l'input e l'output elementare, è necessario includere il file '`stdio.h`', dove, tra l'altro, è dichiarato il tipo '**FILE**'.

67.2.1 Apertura e chiusura

«

L'apertura dei file viene ottenuta normalmente con la funzione ***fopen()*** che restituisce il puntatore al file, oppure il puntatore nullo, '**NULL**', in caso di fallimento dell'operazione. L'esempio seguen-

te mostra l'apertura del file '`mio_file`' contenuto nella directory corrente, con una modalità di accesso in sola lettura.

```
#include <stdio.h>
…
int main (void)
  {
    FILE *fp_mio_file;
    …
    fp_mio_file = fopen ("mio_file", "r");
    …
  }
```

Come si vede dall'esempio, è normale assegnare il puntatore ottenuto a una variabile adatta, che da quel momento identifica il file, finché questo resta aperto.

La chiusura del file avviene in modo analogo, attraverso la funzione ***fclose()***, che restituisce zero se l'operazione è stata conclusa con successo, oppure il valore rappresentato da ***EOF***. L'esempio seguente ne mostra l'utilizzo.

```
…
    fclose (fp_mio_file);
…
```

La chiusura del file conclude l'attività con questo, dopo avere scritto tutti i dati eventualmente ancora rimasti in sospeso (se il file è stato aperto in scrittura).

Normalmente, un file aperto viene definito come flusso di file, o *stream*; così, nello stesso modo viene identificata la variabile puntatore che vi si riferisce. In effetti, lo stesso file potrebbe anche essere aperto più volte con puntatori differenti, quindi è corretto distinguere tra file fisici su disco e file aperti, o flussi.

Seguono gli schemi sintattici di ***fopen()*** e ***fclose()***, in forma di prototipo di funzione:

```
FILE *fopen (char *file, char *modalità);
```

```
int fclose (FILE *flusso_di_file);
```

La funzione ***fopen()*** richiede come secondo argomento una stringa contenente l'informazione della modalità di accesso. Questa può essere composta utilizzando i simboli seguenti, dove la lettera '**b**' richiede espressamente un accesso binario, mentre la mancanza di tale lettera indica un accesso con le convenzioni dei file di testo:

Stringa	Descrizione
`r` `rb`	apre il file in sola lettura, posizionandosi all'inizio del file;
`r+` `rb+` | `r+b`	apre il file in lettura e scrittura, posizionandosi all'inizio del file;

Stringa	Descrizione
`w` `wb`	apre il file in sola scrittura, creandolo se necessario, o troncandone a zero il suo contenuto se questo esiste già;
`w+` `wb+` | `w+b`	apre il file in scrittura e lettura, creandolo se necessario, o troncandone a zero il suo contenuto se questo esiste già;
`a` `ab`	apre il file in scrittura in aggiunta (*append*), creandolo se necessario, o aggiungendovi dati a partire dalla fine e, di conseguenza, posizionandosi alla fine dello stesso;
`a+` `ab+` | `a+b`	apre il file in scrittura in aggiunta e in lettura, creandolo se necessario, o aggiungendovi dati a partire dalla fine e, di conseguenza, posizionandosi alla fine dello stesso.

La funzione ***fclose()*** restituisce zero in caso di successo, oppure il valore corrispondente alla macro-variabile ***EOF*** (annotando anche un valore appropriato nella variabile ***errno***).

67.2.2 Lettura e scrittura

L'accesso al contenuto dei file avviene generalmente a livello di byte e le operazioni di lettura e scrittura dipendono da un indicatore riferito a una posizione, espressa in byte, del contenuto del file stesso. Naturalmente, tale indicatore fa parte delle informazioni che si conservano nella variabile strutturata di tipo '**`FILE`**', a cui si fa riferimento per identificare il flusso di file.

A seconda di come viene aperto il file, questo indicatore viene posizionato nel modo più logico, come descritto a proposito della fun-

zione ***fopen()***. Questo indicatore viene spostato automaticamente a seconda delle operazioni di lettura e scrittura che si compiono, tuttavia, quando si passa da una modalità di accesso all'altra, è necessario spostare l'indicatore attraverso le istruzioni opportune, in modo da non creare ambiguità.

Per la lettura generica di un file in modo binario (nel senso di una lettura tale e quale del file) si può usare la funzione ***fread()*** che legge una quantità di byte trattandoli come un array. Per la precisione, si tratta di definire la dimensione di ogni elemento, espressa in byte, quindi la quantità di tali elementi. Il risultato della lettura viene inserito in un array, i cui elementi hanno la stessa dimensione. Si osservi l'esempio seguente:

```
…
    char ca[100];
    FILE *fp;
    int i;
    …
    i = fread (ca, 1, 100, fp);
    …
```

In questo modo si intende leggere 100 elementi della dimensione di un solo byte, collocandoli nell'array ***ca***, organizzato nello stesso modo. Naturalmente, non è detto che la lettura abbia successo, o quantomeno non è detto che si riesca a leggere la quantità di elementi richiesta. Il valore restituito dalla funzione rappresenta la quantità di elementi letti effettivamente. Se si verifica un qualsiasi tipo di errore che impedisce la lettura, la funzione si limita a restituire zero.

Quando il file viene aperto in lettura, l'indicatore interno viene posizionato all'inizio del file; quindi, ogni operazione di lettura sposta

in avanti il puntatore, in modo che la lettura successiva avvenga a partire dalla posizione immediatamente seguente:

```
…
     char ca[100];
     FILE *fp;
     int i;
     …
     fp = fopen ("mio_file", "rb");
     …
     while (1)          /* Ciclo senza fine */
       {
         i = fread (ca, 1, 100, fp);
         if (i == 0)
           {
             break;       /* Termina il ciclo */
           }
         …
       }
     …
```

In questo modo, come mostra l'esempio, viene letto tutto il file a colpi di 100 byte alla volta, tranne l'ultima in cui si ottiene solo quello che resta da leggere.

Analogamente, la scrittura può essere eseguita con la funzione ***fwrite()*** che scrive una quantità di byte trattandoli come un array, nello stesso modo già visto con la funzione ***fread()***. La scrittura procede a partire dalla posizione corrente riferita al file.

```
…
    char ca[100];
    FILE *fp;
    int i;
    …
    i = fwrite (ca, 1, 100, fp);
    …
```

L'esempio, come nel caso di ***fread()***, mostra la scrittura di 100 elementi di un solo byte, prelevati da un array. Il valore restituito dalla funzione è la quantità di elementi che sono stati scritti con successo. Se si verifica un qualsiasi tipo di errore che impedisce la scrittura, la funzione si limita a restituire zero.

Anche in scrittura è importante l'indicatore della posizione interna del file. Di solito, quando si crea un file o lo si estende, l'indicatore si trova sempre alla fine. L'esempio seguente mostra lo scheletro di un programma che crea un file, copiando il contenuto di un altro (non viene utilizzato alcun tipo di controllo degli errori).

```
#include <stdio.h>
…
int main (void)
{
    char ca[1024];
    FILE *fp_in;
    FILE *fp_out;
    int i;
    …
    fp_in = fopen ("file_in", "r");
    …
    fp_out = fopen ("file_out", "w");
    …
    while (1)                    // Ciclo senza fine.
```

```
        {
          i = fread (ca, 1, 1024, fp_in);
          if (i == 0)
            {
              break;                    // Termina il ciclo.
            }
          …
          fwrite (ca, 1, i, fp_out);
          …
        }
      …
      fclose (fp_in);
      fclose (fp_out);
      …
      return 0;
}
```

Seguono i modelli sintattici di ***fread()*** e ***fwrite()***, espressi in forma di prototipi di funzione:

```
size_t fread (void *restrict ptr,
              size_t dimensione,
              size_t quantità,
              FILE *restrict stream);
```

```
size_t fwrite (const void *restrict ptr,
               size_t dimensione,
               size_t quantità,
               FILE *stream);
```

Il tipo di dati '**`size_t`**' serve a garantire la compatibilità con qualun-

que tipo intero, mentre il tipo '**void**' per l'array permette l'utilizzo di qualunque tipo per i suoi elementi, anche se negli esempi è sempre stato visto il trattamento di sole sequenze di byte.

67.2.3 Indicatore interno al file

«

Lo spostamento diretto dell'indicatore interno della posizione di un file aperto è un'operazione necessaria quando il file è stato aperto simultaneamente in lettura e in scrittura, e da un tipo di operazione si vuole passare all'altro. Per questo si utilizza la funzione ***fseek()*** ed eventualmente anche ***ftell()*** per conoscere la posizione attuale. La posizione e gli spostamenti sono espressi in byte.

La funzione ***fseek()*** esegue lo spostamento a partire dall'inizio del file, oppure dalla posizione attuale, oppure dalla posizione finale. Per questo utilizza un parametro che può avere tre valori identificati rispettivamente da tre macro-variabili: ***SEEK_SET***, ***SEEK_CUR*** e ***SEEK_END***. l'esempio seguente mostra lo spostamento del puntatore, riferito al flusso di file ***fp***, in avanti di 10 byte, a partire dalla posizione attuale.

```
…
i = fseek (fp, 10, SEEK_CUR);
…
```

La funzione ***fseek()*** restituisce zero se lo spostamento avviene con successo, altrimenti si ottiene un valore negativo.

L'esempio seguente mostra lo scheletro di un programma, senza controlli sugli errori, che, dopo aver aperto un file in lettura e scrittura, lo legge a blocchi di dimensioni uguali, modifica questi blocchi e li riscrive nel file.

```
#include <stdio.h>

static const int dim = 100; // Dimensione del record logico.

int main (void)
{
    char ca[dim];
    FILE *fp;
    int qta;
    int posizione_1;
    int posizione_2;

    fp = fopen ("mio_file", "r+b");  // Lettura e scrittura.

    while (1)                        // Ciclo senza fine.
      {
        //
        // Salva la posizione del puntatore interno al file
        // prima di eseguire la lettura.
        //
        posizione_1 = ftell (fp);
        qta = fread (ca, 1, dim, fp);

        if (qta == 0)
          {
            break;                   // Termina il ciclo.
          }
        //
        // Salva la posizione del puntatore interno al file
        // dopo la lettura.
        //
        posizione_2 = ftell (fp);
        //
        // Sposta il puntatore alla posizione precedente
```

```
            // alla lettura.
            //
            fseek (fp, posizione_1, SEEK_SET);
            //
            // Esegue qualche modifica nei dati, per esempio
            // mette un punto esclamativo all'inizio.
            //
            ca[0] = '!';
            //
            // Riscrive il record modificato.
            //
            fwrite (ca, 1, qta, fp);
            //
            // Riporta il puntatore interno al file alla
            // posizione corretta per eseguire la lettura
            // successiva.
            //
            fseek (fp, posizione_2, SEEK_SET);
          }

        fclose (fp);
        return 0;
}
```

Segue il modello sintattico per l'uso della funzione ***fseek()***, espresso attraverso il suo prototipo:

```
int fseek (FILE *stream, long int spostamento,
           int punto_di_partenza);
```

Il valore dello spostamento, fornito come secondo parametro, rappresenta una quantità di byte che può essere anche negativa, indicando in tal caso un arretramento dal punto di partenza. Il valore restitui-

to da ***fseek()*** è zero se l'operazione viene completata con successo, altrimenti viene restituito un valore diverso.

Segue il modello sintattico per l'uso della funzione ***ftell()***, espresso attraverso il suo prototipo:

```
long int ftell (FILE *stream)
```

La funzione ***ftell()*** permette di conoscere la posizione dell'indicatore interno al file a cui fa riferimento il flusso di file fornito come parametro. Se si tratta di un file per il quale si esegue un accesso binario, la posizione ottenuta è assoluta, ovvero riferita all'inizio del file.

Il valore restituito in caso di successo è positivo, a indicare appunto la posizione dell'indicatore. Se si verifica un errore viene restituito un valore negativo: -1.

67.2.4 File di testo

I file di testo possono essere gestiti in modo più semplice attraverso due funzioni: ***fgets()*** e ***fputs()***. Queste permettono rispettivamente di leggere e scrivere un file una riga alla volta, intendendo come riga una porzione di testo che termina con il codice di interruzione di riga, secondo l'astrazione usata dal linguaggio.

La funzione ***fgets()*** permette di leggere una riga di testo di una data dimensione massima. Si osservi l'esempio seguente:

```
…
fgets (ca, 100, fp);
…
```

In questo caso, viene letta una riga di testo di una dimensione massima di 99 caratteri, dal file rappresentato dal puntatore ***fp***. Questa riga viene posta all'interno dell'array ***ca***, con l'aggiunta di un carattere '`\0`' finale. Questo fatto spiega il motivo per il quale il secondo parametro corrisponde a 100, mentre la dimensione massima della riga letta è di 99 caratteri. In pratica, l'array di destinazione è sempre una stringa, terminata correttamente.

Nello stesso modo funziona ***fputs()***, che però richiede solo la stringa e il puntatore del file da scrivere. Dal momento che una stringa contiene già l'informazione della sua lunghezza perché possiede un carattere di conclusione, non è prevista l'indicazione della quantità di elementi da scrivere.

```
…
fputs (ca, fp);
…
```

Seguono i modelli sintattici delle funzioni ***fputs()*** e ***fgets()***, in forma di prototipi di funzione:

```
char *fgets (char *stringa, int dimensione_max, FILE *stream);
```

```
int fputs (const char *stringa, FILE *stream)
```

Se l'operazione di lettura riesce, ***fgets()*** restituisce un puntatore corrispondente alla stessa stringa (cioè l'array di caratteri di destinazione), altrimenti restituisce il puntatore nullo, '**`NULL`**', per esempio quando è già stata raggiunta la fine del file.

La funzione ***fputs()*** permette di scrivere una stringa in un file di

testo. La stringa viene scritta senza il codice di terminazione finale, '`\0`', ma anche senza aggiungere il codice di interruzione di riga. Il valore restituito è un valore positivo in caso si successo, altrimenti ***EOF***.

In alternativa a ***fgets()*** e a ***fputs()*** si possono considerare anche le funzioni ***gets()*** e ***puts()***, le quali però utilizzano rispettivamente lo standard input e lo standard output. Ma la funzione ***gets()*** legge tutto quello che trova fino alla fine della riga o, in mancanza di questo, fino alla fine del file, mentre ***puts()*** **aggiungere automaticamente il codice di interruzione di riga** alla fine della stringa che viene scritta nel file.

```
char *gets (char *stringa);
```

```
int puts (const char *stringa)
```

67.2.5 I/O standard

Ci sono tre flussi di file che risultano aperti in modo predefinito, all'avvio del programma:

- standard input, corrispondente normalmente alla tastiera;
- standard output, corrispondente normalmente allo schermo del terminale;
- standard error, anch'esso corrispondente normalmente allo schermo del terminale.

Spesso si utilizzano questi flussi di file attraverso funzioni apposite (come nel caso di ***gets()*** e ***puts()***) che vi fanno riferimento in modo

implicito, ma si potrebbe accedere anche attraverso funzioni generalizzate, utilizzando come puntatori i nomi: '**stdio**', '**stdout**' e '**stderr**'.

67.2.6 Ridirezione

«

È possibile associare un flusso di file già in essere, a un file differente, attraverso la funzione ***freopen()***, oppure è possibile modificarne la modalità di accesso. Evidentemente questo tipo di operazione richiede la chiusura del flusso di file, prima di associarvi un file differente o di cambiare la modalità, cosa che comunque tenta di eseguire automaticamente la stessa funzione ***freopen()***:

```
FILE *freopen (const char *restrict nome_file_nuovo,
               const char *restrict modalità_di_accesso,
               FILE *restrict flusso di file);
```

La funzione, se riesce a eseguire il proprio compito, restituisce il puntatore allo stesso flusso di file indicato come terzo argomento, ovvero quello a cui viene applicata la ridirezione o la modifica dei permessi (o entrambe le cose). Per limitare l'effetto alla sola modifica della modalità di accesso, è sufficiente indicare il puntatore nullo al posto del nome del file. Viene mostrato un esempio che ridirige lo standard output:

```
#include <stdio.h>
int main (void)
{
    printf ("ciao 1\n");
    freopen ("mio", "w", stdout);
    printf ("ciao 2\n");
    freopen ("/dev/tty", "w", stdout);
    printf ("ciao 3\n");
    return 0;
}
```

In questo caso, dal momento che la funzione ***printf()*** scrive automaticamente attraverso lo standard output, quando il flusso di file '**`stdout`**' viene ridiretto nel file '`mio`', il testo '**`ciao 2`**' viene scritto in tale file. Ipotizzando di operare in un sistema Unix o in un sistema equivalente, il file di dispositivo '`/dev/tty`' dovrebbe corrispondere allo schermo del terminale utilizzato in quel momento (anche se fosse un terminale grafico); pertanto, il messaggio '**`ciao 3`**' dovrebbe apparire nuovamente sullo schermo.

Logicamente, quando si riapre un file e si cambia la modalità, da binaria a testo o viceversa, può essere appropriato un riposizionamento, con l'aiuto di ***fseek()***.

67.2.7 Controllo degli errori

Molte funzioni, quando si verifica un errore, annotano quanto accaduto, in forma di numero intero, in una variabile globale nota con il nome ***errno***. In generale, il nome ***errno*** è un'espressione che si traduce nell'accesso, a un'area di memoria, condiviso dal programma, ed eventualmente distinto in base al thread, ovvero il flusso di controllo. Il significato del valore attribuito alla variabile ***errno*** è descritto da macro-variabili definite nel file '`errno.h`', nel qua-

le viene anche dichiarata la variabile ***errno***, o l'espressione che la rappresenta.

La lettura della variabile ***errno*** porta alla conoscenza dell'ultimo errore che si è presentato e non è previsto il suo azzeramento automatico.

La variabile strutturata che si utilizza per fare riferimento a un flusso di file prevede anche l'annotazione di uno stato di errore. In pratica, le funzioni che accedono ai file, oltre che aggiornare la variabile globale ***errno***, gestiscono l'indicazione di questo stato, azzerandolo quando non è più significativo. Per verificare la presenza di uno stato di errore ancora valido, a proposito di un flusso di file, si usa la funzione ***ferror()*** che restituisce un valore diverso da zero se questo stato esiste effettivamente:

```
int ferror (FILE *flusso_di_file);
```

Per interpretare l'errore annotato nella variabile ***errno*** e visualizzare direttamente un messaggio attraverso lo standard error, si può usare la funzione ***perror()***:

```
void perror (const char *s);
```

La funzione ***perror()*** mostra un messaggio in modo autonomo, aggiungendo davanti la stringa che può essere fornita come primo argomento (diversamente si può indicare il puntatore nullo o una stringa nulla, in quanto contenente solo il carattere di terminazione).

L'esempio seguente mostra un programma completo e molto semplice, in cui si crea un errore, tentando di scrivere un messaggio

attraverso lo standard input, cosa che produce un errore. Se effettivamente si rileva un errore associato a quel flusso di file, attraverso la funzione ***ferror()***, allora si passa alla sua interpretazione con la funzione ***perror()***.

Listato 67.15. Per provare il codice attraverso un servizio *pastebin*: *http://ideone.com/JvbUI* .

```
#include <stdio.h>
#include <errno.h>
int main (void)
{
    fprintf (stdin, "Ciao amore!\n");
    if (ferror (stdin))
      {
        perror ("Attenzione");
      }
    return 0;
}
```

Come si vede, è necessario includere anche il file ‘`errno.h`’, senza il quale la variabile ***errno*** non risulterebbe accessibile. Avviando questo programma in un sistema GNU/Linux si potrebbe ottenere il messaggio seguente:

```
Attenzione: Bad file descriptor
```

In alternativa alla funzione ***perror()*** si può usare anche ***strerror()*** (dal file ‘`string.h`’), con la quale si ottiene la stringa contenente il messaggio di errore:

```
char *strerror (int n_errore);
```

Si può modificare leggermente l’esempio già apparso, in modo da

usare la funzione ***strerror()*** per produrre lo stesso risultato.

Listato 67.17. Per provare il codice attraverso un servizio *pastebin*: *http://ideone.com/qDCYr* .

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
int main (void)
{
    char *cp;
    fprintf (stdin, "Ciao amore!\n");
    if (ferror (stdin))
      {
        cp = strerror (errno);
        fprintf (stderr, "Attenzione: %s\n", cp);
      }
    return 0;
}
```

67.3 Conversione di input e output

«

Il linguaggio C rappresenta in memoria i valori numerici in modo binario secondo una modalità diversa rispetto a quella usata per le stringhe che servono invece per l'interazione umana. In altri termini, un conto è il valore 100, un altro è la sequenza dei caratteri numerici con cui questo valore viene rappresentato sullo schermo o su carta.

Il linguaggio C non svolge automaticamente conversioni da valori numerici binari a stringhe di cifre numeriche e viceversa; per questo è necessario invece avvalersi di funzioni di conversione. Per la precisione esistono due gruppi di funzioni, ...***printf()*** e ...***scanf()***, con cui è possibile comporre (nel senso tipografico) le informazioni in uscita, oppure interpretarle in senso inverso le informazioni in ingresso.

67.3.1 Composizione dell'output

Le funzioni del gruppo ...***printf()*** consentono di comporre una stringa (da memorizzare o da visualizzare), partendo da un'altra stringa contenente il formato di composizione e utilizzando un elenco variabile di argomenti:

```
...printf ( ... stringa_di_composizione [, argomento] ... )
```

Il modello sintattico dà solo una visione di massima: a seconda della funzione ci possono essere dei parametri che non vengono chiariti nello schema, quindi appare sempre la stringa di composizione, la quale può essere seguita da altri argomenti le cui caratteristiche non sono precisate nel prototipo della funzione.[3]

La stringa di composizione è una stringa normale, in cui si inseriscono delle sequenze precedute dal simbolo '`%`', note come ***specificatori di conversione***. Conviene partire da un esempio, proprio con la funzione ***printf()***, la quale emette la stringa generata dalla composizione attraverso lo standard output (attraverso il flusso di file associato allo standard output):

```
...
printf ("Il capitale di %i al tasso %f%% dà l'interesse %i",
        1000, 0.5, 5);
...
```

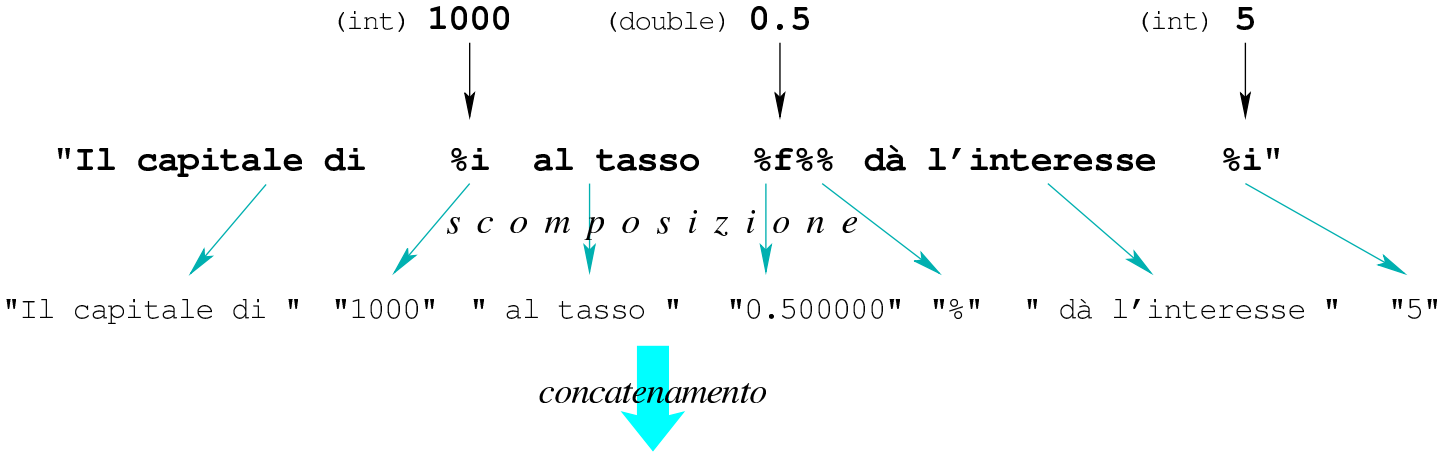
Da questa istruzione si ottiene la visualizzazione della frase seguente:

```
Il capitale di 1000 al tasso 0.500000% dà l'interesse 5
```

In pratica, al posto del primo specificatore '`%i`' è stato inserito il va-

lore 1000 dopo averlo convertito in modo da essere rappresentato da quattro caratteri (‘`'1'`’, ‘`'0'`’, ‘`'0'`’, ‘`'0'`’), al posto del secondo specificatore ‘`%f`’ è stato inserito il valore 0.5 dopo un’opportuna conversione in caratteri, al posto del terzo specificatore ‘`%%`’ è stato inserito un carattere di percentuale, infine, al posto del quarto specificatore ‘`%i`’ è stato inserito il valore 5.

Figura 67.20. Schematizzazione della trasformazione di una stringa di composizione in una stringa finale.



Il capitale di 1000 al tasso di 0.500000% dà l’interesse 5

Lo specificatore di conversione ha due compiti: indicare che tipo di informazione viene prelevato dagli argomenti (ammesso che si prelevi effettivamente un valore) e come questa deve essere rappresentata. Nel caso dell’esempio, il primo specificatore ‘`%i`’ indica che il valore da prelevare dagli argomenti è di tipo ‘`int`’; il secondo specificatore ‘`%f`’ indica un tipo ‘`double`’; il terzo non preleva alcun valore; il quarto indica ancora un altro ‘`int`’.

Una stringa di composizione che non contenga degli specificatori rimane evidentemente intatta e non richiede alcun dato aggiuntivo. La funzione ***printf()*** (che è stata usata nell’esempio) viene usata spesso come mezzo generico per emettere un messaggio attraverso lo stan-

dard output, anche quando non c'è alcun bisogno di comporre dei dati. Questo è lecito, ma non va dimenticato il contesto, pertanto, scrivere l'istruzione seguente sarebbe sbagliato:

```
…
printf ("Il capitale di 1000 al tasso 0.5% dà l'interesse 5");
…
```

Il modo giusto è quello seguente:

```
…
printf ("Il capitale di 1000 al tasso 0.5%% dà l'interesse 5");
…
```

67.3.2 Rappresentazione degli specificatori di composizione per l'emissione dei dati

Di norma, la scelta dello specificatore determina il tipo di dati dell'argomento e il tipo di trasformazione che deve ricevere. La tabella 67.23 elenca alcuni degli specificatori di conversione utilizzabili, nella loro forma più semplice. È bene ricordare che per rappresentare il simbolo di percentuale si usa uno specificatore fittizio composto dalla sequenza di due segni percentuali: '`%%`'.

Tabella 67.23. Alcuni specificatori di conversione.

Simbolo	Corrispondenza
`%…c`	Un carattere singolo.
`%…s`	Una stringa.
`%…d`	Un intero con segno da rappresentare in base dieci.

Simbolo	Corrispondenza
`%...u`	Un intero senza segno da rappresentare in base dieci.
`%...o`	Un intero senza segno da rappresentare in ottale.
`%...x`	Un intero senza segno da rappresentare in esadecimale.
`%...e`	Un numero a virgola mobile normale (‘**`double`**’), da rappresentare in notazione esponenziale.
`%...f`	Un numero a virgola mobile normale (‘**`double`**’), da rappresentare in notazione decimale fissa.

Leggendo la tabella si può osservare che la composizione dei dati in uscita può riguardare anche dati che sono già in forma di stringa (lo specificatore ‘**`%...s`**’), pertanto si usa questo metodo anche per il concatenamento delle stringhe.

Gli specificatori di conversione possono contenere indicazioni ulteriori tra il simbolo di percentuale e la lettera che definisce il tipo di trasformazione. Si tratta di inserire un simbolo composto da un carattere singolo, seguito eventualmente da altre informazioni aggiuntive, secondo la sintassi seguente:

```
% [simbolo] [n_ampiezza] [.n_precisione] [hh|h|l|ll|j|z|t|L] tipo
```

Alcuni di questi simboli sono rappresentati dalla tabella 67.24. In presenza di valori numerici, si può indicare il numero di cifre decimali intere (ampiezza), aggiungendo eventualmente il numero di decimali (precisione), se si tratta di rappresentare un numero a virgola mobile. Quando è necessario modificare il tipo di dati provenienti dagli argomenti, ciò può essere precisato con una sigla, come

descritto nella tabella 67.25.

Tabella 67.24. Alcuni simboli per la conversione di valori numerici.

Simbolo	Corrispondenza
`%+`...	Il segno «+», usato all'inizio di uno specificatore di conversione, fa sì che i numeri positivi siano rappresentati con il segno in modo esplicito, mentre altrimenti il segno viene mostrato solo per quelli negativi.
`%0`***ampiezza***... `%+0`***ampiezza***...	Lo zero, usato all'inizio di uno specificatore di conversione, fa sì che si inseriscano degli zeri per allineare a destra un valore numerico, nell'ambito dell'ampiezza specificata. Lo zero può combinarsi con il segno «+».
`%`***ampiezza***... `%+`***ampiezza***...	In mancanza di uno zero iniziale, in presenza dell'indicazione dell'ampiezza, il valore viene allineato a destra usando degli spazi.
`%-`... `%-+`...	Il segno meno richiede un allineamento a sinistra rispetto al campo, usando degli spazi a destra. Si può combinare con il segno «+», ma non con lo zero.

Tabella 67.25. Alcuni modificatori dell'estensione che ha in memoria il valore da estrarre e comporre.

Simbolo	Corrispondenza

`%...ld`
`%...lu`
`%...lo`
`%...lx`
	La lettera '**`l`**', prima di quella che specifica il tipo di composizione, quando si tratta della conversione di un numero intero, specifica che il dato va trattato in qualità di '**`long int`**', con o senza segno.

Simbolo	Corrispondenza
`%…lc` `%…ls`	La lettera '**`l`**', prima di quella che specifica il tipo di composizione, quando si tratta della conversione di un carattere o di una stringa, specifica che il dato va trattato in qualità di carattere esteso (*wide char*) o di stringa estesa (*wide string*).

`%…lld`
`%…llu`
`%…llo`
`%…llx`
	La coppia di lettere '**`ll`**', prima di quella che specifica il tipo di composizione, quando si tratta della conversione di un numero intero, specifica che il dato va trattato in qualità di '**`long long int`**', con o senza segno.

`%…Le` `%…Lf`	La lettera '**`L`**', prima di quella che specifica il tipo di composizione, quando si tratta della conversione di un valore in virgola mobile, specifica che il dato va trattato in qualità di '**`long double`**'.

Nella stringa di composizione possono apparire anche sequenze di escape come già mostrato nella tabella 66.17. Si veda anche la pagina di manuale *printf(3)*.

Tabella 67.26. Esempi di utilizzo degli specificatori di conversione di ***printf()***. Le costanti numeriche utilizzate negli esempi sono interpretate secondo le convenzioni del linguaggio, pertanto: 123 e −123 sono costanti di tipo '**`int`**'; mentre 123.456 e −123.456 sono costanti di tipo '**`double`**'.

Codice	Risultato emesso attraverso la funzione
`printf ("[%i]", 123);`	`[123]`
`printf ("[%i]", -123);`	`[-123]`

Codice	Risultato emesso attraverso la funzione
`printf ("[%2d]", 123);`	`[123]` L'indicatore '**`%2d`**' specifica che si devono usare almeno due cifre, ma se le cifre della parte intera sono in numero maggiore, queste vanno indicate tutte ugualmente.
`printf ("[%6d]", 123);`	`[   123]`
`printf ("[%6d]", -123);`	`[  -123]`
`printf ("[%+6d]", 123);`	`[  +123]`
`printf ("[%06d]", 123);`	`[000123]`
`printf ("[%06d]", -123);`	`[-00123]`
`printf ("[%+06d]", 123);`	`[+00123]`
`printf ("[%-6d]", 123);`	`[123   ]`
`printf ("[%u]", 123);`	`[123]`
`printf ("[%u]", -123);`	`[4294967173]` Evidentemente si ottiene la rappresentazione del valore binario, tale e quale, secondo la notazione usata per i valori negativi.
`printf ("[%6x]", 123);`	`[    7b]`
`printf ("[%06x]", 123);`	`[00007b]`
`printf ("[%x]", 123);`	`[7b]`

Codice	Risultato emesso attraverso la funzione
`printf ("[%x]", -123);`	`[ffffff85]` Evidentemente si ottiene la rappresentazione del valore binario, tale e quale, secondo la notazione usata per i valori negativi.
`printf ("[%6x]", 123);`	`[    7b]`
`printf ("[%06x]", 123);`	`[00007b]`
`printf ("[%o]", 123);`	`[173]`
`printf ("[%o]", -123);`	`[37777777605]` Evidentemente si ottiene la rappresentazione del valore binario, tale e quale, secondo la notazione usata per i valori negativi.
`printf ("[%6o]", 123);`	`[   173]`
`printf ("[%06o]", 123);`	`[000173]`
`printf ("[%f]", 123.456);`	`[123.456000]`
`printf ("[%f]", -123.456);`	`[-123.456000]`
`printf ("[%12f]", 123.456);`	`[  123.456000]`
`printf ("[%.4f]", 123.456);`	`[123.4560]`
`printf ("[%12.4f]", 123.456);`	`[    123.4560]`
`printf ("[%12.4f]", -123.456);`	`[   -123.4560]`
`printf ("[%+12.4f]", 123.456);`	`[   +123.4560]`
`printf ("[%012.4f]", 123.456);`	`[0000123.4560]`
`printf ("[%012.4f]", -123.456);`	`[-000123.4560]`
`printf ("[%+012.4f]", 123.456);`	`[+000123.4560]`

Codice	Risultato emesso attraverso la funzione
`printf ("[%-12.4f]", 123.456);`	`[123.4560    ]`
`printf ("[%e]", 123.456);`	`[1.234560e+02]`
`printf ("[%e]", -123.456);`	`[-1.234560e+02]`
`printf ("[%15e]", 123.456);`	`[   1.234560e+02]`
`printf ("[%.4e]", 123.456);`	`[1.2346e+02]`
`printf ("[%15.4e]", 123.456);`	`[     1.2346e+02]`
`printf ("[%15.4e]", -123.456);`	`[    -1.2346e+02]`
`printf ("[%+15.4e]", 123.456);`	`[    +1.2346e+02]`
`printf ("[%015.4e]", 123.456);`	`[000001.2346e+02]`
`printf ("[%015.4e]", -123.456);`	`[-00001.2346e+02]`
`printf ("[%+015.4e]", 123.456);`	`[+00001.2346e+02]`
`printf ("[%-15.4e]", 123.456);`	`[1.2346e+02     ]`
`printf ("[%s]", "ciao amore");`	`[ciao amore]`
`printf ("[%7s]", "ciao amore");`	`[ciao amore]` La stringa è più lunga di sette caratteri, ma viene visualizzata completamente.
`printf ("[%.7s]", "ciao amore");`	`[ciao am]` La stringa viene troncata se è più lunga del valore della precisione.
`printf ("[%.14s]", "ciao amore");`	`[ciao amore]`
`printf ("[%14s]", "ciao amore");`	`[    ciao amore]`
`printf ("[%14.7s]", "ciao amore");`	`[       ciao am]`
`printf ("[%-14s]", "ciao amore");`	`[ciao amore    ]`

Codice	Risultato emesso attraverso la funzione
`printf ("[%-14.7s]", "ciao amore");`	`[ciao am       ]`

67.3.3 Funzioni per la composizione dell'output

Tutte le funzioni standard il cui nome finisce per '**`printf`**' interpretano una stringa di composizione secondo le modalità descritte nel capitolo, ovvero in modo analogo a ***printf()*** che, in particolare, emette il risultato della composizione attraverso lo standard output. In particolare, la funzione ***fprintf()*** scrive il risultato attraverso il flusso di file che costituisce il parametro ***stream*** (il primo argomento) e la funzione ***sprintf()*** copia il risultato, come stringa, a partire dal puntatore ***s*** (sempre il primo argomento).

```
int printf (const char *restrict composizione, ...);
```

```
int fprintf (FILE *restrict stream,
             const char *restrict composizione, ...);
```

```
int sprintf (char *restrict s,
             const char *restrict composizione, ...);
```

```
int snprintf (char *restrict s,
              size_t n,
              const char *restrict composizione, ...);
```

Le funzioni di cui è appena stato mostrato il modello sintattico, leggono gli argomenti successivi alla stringa di composizione in base a quanto indicato con gli specificatori di composizione. Altre funzioni equivalenti, con il nome che inizia con la lettera «v», hanno bisogno di un puntatore di tipo '**va_list**':

```
int vprintf (const char *restrict composizione,
             va_list arg);
```

```
int vfprintf (FILE *restrict stream,
              const char *restrict composizione,
              va_list arg);
```

```
int vsprintf (char *restrict s,
              const char *restrict composizione,
              va_list arg);
```

```
int vsnprintf (char *restrict s,
               size_t n,
               const char *restrict composizione,
               va_list arg);
```

67.3.4 Concatenamento di stringhe

Il linguaggio C, di per sé, non agevola l'uso delle stringhe; al massimo si può contare sul fatto che una sequenza di stringhe letterali venga considerata una stringa sola, concatenata. Per il concatenamento

delle stringhe sono disponibili le funzioni ***strcat()*** e ***strncat()***, ma l'uso delle funzioni previste per la composizione dell'output è molto più comodo, considerata la facilità con cui si inseriscono anche dati diversi dalle stringhe.

Listato 67.27. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/Nzbovidr* , *http://ideone.com/Jgn8z* .

```
#include <stdio.h>
int main (void)
{
    char s[] = "Ciao amore";
    printf ("%s... Ti %s.\n", s, "voglio tanto bene");
    return 0;
}
```

L'esempio appena mostrato dovrebbe dimostrare questa maggiore facilità. Il messaggio che viene visualizzato è: «Ciao amore... Ti voglio tanto bene.»

67.3.5 Interpretazione dell'input

«

Quando un programma interagisce con l'essere umano, scambia dati in forma grafica, nel senso che un numero appare e viene inserito come sequenza di caratteri grafici. Così come per la rappresentazione umana dei dati si usano comunemente le funzioni ...***printf()***, per l'immissione dei dati si usano le funzioni ...***scanf()*** che hanno il ruolo opposto:

```
...scanf ( ... stringa_di_conversione [ , argomento ] ... )
```

Il modello sintattico dà solo una visione di massima: a seconda della

funzione ci possono essere dei parametri che non vengono chiariti nello schema, quindi appare sempre la stringa di conversione, la quale può essere seguita da altri argomenti costituiti da puntatori, le cui caratteristiche particolari non sono precisate nel prototipo della funzione.[4] Viene proposto un esempio con la funzione ***scanf()*** che riceve i dati in ingresso (da interpretare) dallo standard input:

```
…
printf ("Inserisci l'importo: ");
scanf ("%i", &i_importo);
…
```

Il pezzo di codice mostrato emette la frase seguente e resta in attesa dell'inserimento di un valore numerico intero, seguito da [*Invio*]:

```
Inserisci l'importo: _
```

Questo valore viene inserito nella variabile ***i_importo***. Si deve osservare il fatto che i parametri successivi alla stringa di conversione sono dei puntatori, per cui, avendo voluto inserire il dato nella variabile ***i_importo***, questa è stata indicata preceduta dall'operatore '**&**' in modo da fornire alla funzione l'indirizzo corrispondente (si veda la sezione 66.5 sulla gestione dei puntatori).

Con una stessa funzione di questo tipo è possibile inserire dati per diverse variabili, come si può osservare dall'esempio seguente, ma in questo caso, per ogni dato viene richiesta la pressione di [*Invio*] o l'inserimento di spazi tra un dato e l'altro.

```
…
printf ("Inserisci il capitale e il tasso: ");
scanf ("%i%f", &i_capitale, &i_tasso);
…
```

La stringa di conversione è il parametro più delicato di queste funzioni. Come visto negli esempi, una stringa del genere contiene principalmente degli specificatori di conversione che, come già accennato, si comportano in modo molto simile agli specificatori di composizione delle funzioni *...printf()*. Quello che segue è lo schema sintattico generale per la definizione di uno specificatore di conversione:

```
%[*] [n_ampiezza] [hh|h|l|ll|j|z|t|L] tipo
```

Come si può vedere, all'inizio è previsto un solo tipo di simbolo, costituito da un asterisco, il cui scopo è quello di annullare l'assegnamento del valore a una variabile. In pratica, con l'asterisco il dato corrispondente allo specificatore viene interpretato, ma poi non viene salvato in alcuna variabile.

Successivamente può apparire un numero che rappresenta l'ampiezza del dato da interpretare, in byte, il cui scopo è quello di limitare la lettura fino a un certo carattere (inteso come '**char**', pertanto le sequenze multibyte contano per più di una unità singola). In questo caso non esiste la possibilità di indicare una precisione.

Dopo può apparire una sigla, composta da una o più lettere, il cui scopo è quello di modificare la dimensione predefinita della variabile di destinazione. In altri termini, senza questo modificatore si intende che la variabile ricevente debba essere di una certa grandezza, ma con l'aggiunta del «modificatore di lunghezza» si precisa invece qualcosa di diverso. In pratica, il modificatore di lunghezza usato da queste funzioni è equivalente a quello delle funzioni di composizione dell'output.

Al termine dello specificatore di conversione appare una lettera che

dichiara come deve essere interpretato il dato in ingresso e, in mancanza del modificatore di lunghezza, indica anche la dimensione predefinita della variabile ricevente.

Secondo la documentazione standard, il contenuto delle stringhe di conversione si suddivide in «direttive» che, in linea di massima, dovrebbero comporsi secondo il modello seguente:

[*spazi*] *carattere_multibyte* | %...

Pertanto, una direttiva può contenere degli spazi, un carattere (inteso in senso tipografico e quindi può occupare più di un byte) oppure uno specificatore di conversione. Visto da un altro punto di vista, la stringa di conversione è composta principalmente da specificatori di conversione che però possono essere alternati da spazi o altri caratteri: gli spazi indicano che in quella posizione possono esserci spazi che vengono ignorati; altri caratteri devono invece corrispondere esattamente nell'input e vengono poi ignorati. Tuttavia ci sono altre situazioni in cui gli spazi sono ugualmente esclusi in modo predefinito, come nell'esempio già visto, dove la stringa di conversione è composta solo da specificatori di conversione. Nell'esempio seguente, invece, si dimostra l'uso di caratteri estranei agli specificatori di conversione:

```
…
printf ("Inserisci la data: ");
scanf ("%i/%i/%i", &giorno, &mese, &anno);
…
```

In questo caso la digitazione della data richiede anche l'inserzione delle barre oblique, senza le quali il riconoscimento fallisce.

Purtroppo, la sintassi per la scrittura delle stringhe di conversione non è molto soddisfacente ed è difficile avere un'idea chiara del loro utilizzo. Pertanto, è consigliabile di utilizzare sempre solo modelli molto semplici.

67.3.6 Rappresentazione degli specificatori di conversione

«

Di norma, la scelta dello specificatore di conversione determina il tipo di dati dell'argomento (ovvero il tipo di variabile a cui l'argomento punta) e il modo in cui deve essere interpretato. La tabella successiva elenca alcuni degli specificatori di conversione utilizzabili, nella loro forma più semplice. È bene ricordare che anche in questo caso si può usare uno specificatore costituito dall'unione di due caratteri percentuali ('`%%`'), il quale identifica semplicemente un carattere di percentuale singolo proveniente dai dati in ingresso, ma da ignorare.

Tabella 67.32. Tipi di conversione principali.

Simbolo	Corrispondenza
%...c	Corrisponde a un carattere, oppure, se viene specificata una lunghezza, a una sequenza di caratteri; pertanto, in condizioni normali il puntatore deve essere di tipo '**`char *`**' e, a partire dalla posizione indicata dallo stesso, ci deve essere abbastanza spazio per contenere tutti i caratteri previsti per l'immissione.

Simbolo	Corrispondenza
`%…s`	Corrisponde a una sequenza di caratteri diversi da quelli che producono uno spazio e a questa sequenza viene aggiunto automaticamente il carattere *<NUL>*, ovvero '**`\0`**'; pertanto, in condizioni normali il puntatore deve essere di tipo '**`char *`**' e, a partire dalla posizione indicata dallo stesso, ci deve essere abbastanza spazio per contenere la stringa da immettere, incluso il carattere nullo conclusivo.
`%…d`	Corrisponde a un numero intero (con o senza segno), espresso in base dieci; pertanto, in condizioni normali il puntatore deve essere di tipo '**`int *`**', con o senza segno.
`%…i`	Corrisponde a un numero intero (con o senza segno), espresso in base otto, in base dieci o in base sedici (purché la base di numerazione sia riconoscibile dal prefisso usato); pertanto, in condizioni normali il puntatore deve essere di tipo '**`int *`**', con o senza segno.
`%…u`	Corrisponde a un numero intero senza segno, espresso in base dieci; pertanto, in condizioni normali il puntatore deve essere di tipo '**`unsigned int *`**'.
`%…x`	Corrisponde a un numero intero senza segno, espresso in base sedici; pertanto, in condizioni normali il puntatore deve essere di tipo '**`unsigned int *`**'.
`%…o`	Corrisponde a un numero intero senza segno, espresso in base otto; pertanto, in condizioni normali il puntatore deve essere di tipo '**`unsigned int *`**'.

Simbolo	Corrispondenza

`%…e`
`%…f`
`%…g`
	Corrisponde a un numero a virgola mobile rappresentato in notazione decimale fissa o in notazione esponenziale; pertanto, in condizioni normali il puntatore deve essere di tipo '**`double *`**'.

Tabella 67.33. Alcuni modificatori dell'ampiezza del valore da immettere.

Simbolo	Corrispondenza

`%…hd`
`%…hi`
`%…hu`
`%…ho`
`%…hx`
	La lettera '**`h`**', prima di quella che specifica il tipo di conversione, quando si tratta dell'inserimento di un numero intero, specifica che il dato va trattato in qualità di '**`short int *`**', con o senza segno, in base al contesto.

`%…ld`
`%…lu`
`%…lo`
`%…lx`
	La lettera '**`l`**', prima di quella che specifica il tipo di conversione, quando si tratta dell'inserimento di un numero intero, specifica che il dato va trattato in qualità di '**`long int *`**', con o senza segno, in base al contesto.

Simbolo	Corrispondenza
`%…lc` `%…ls`	La lettera '**`l`**', prima di quella che specifica il tipo di conversione, quando si tratta dell'inserimento di un carattere o di una stringa, specifica che il dato va trattato in qualità di carattere esteso (*wide char*) o di stringa estesa (*wide string*). Di conseguenza, si intende che l'argomento sia di tipo '**`wchar_t *`**'.

`%…Le`
`%…Lf`
`%…Lg`
	La lettera '**`L`**', prima di quella che specifica il tipo di conversione, quando si tratta dell'inserimento di un valore in virgola mobile, specifica che il dato va trattato in qualità di '**`long double *`**'.

67.3.7 Funzioni per l'interpretazione dell'input

Tutte le funzioni standard il cui nome finisce per '**`scanf`**' interpretano dei dati in ingresso attraverso una stringa di conversione, secondo le modalità descritte nel capitolo, ovvero in modo analogo a ***scanf()*** che, in particolare, legge i dati da interpretare dallo standard input. In particolare, la funzione ***fscanf()*** legge l'input attraverso il flusso di file che costituisce il parametro ***stream*** (il primo argomento) e la funzione ***sscanf()*** legge l'input da una stringa (che costituisce sempre il primo argomento).

```
int fscanf (FILE *restrict stream,
            const char *restrict conversione, ...);
```

```
int sscanf (const char *restrict s,
            const char *restrict conversione, ...);
```

```
int scanf  (const char *restrict conversione, ...);
```

Le funzioni di cui è appena stato mostrato il modello sintattico, utilizzano gli argomenti successivi alla stringa di conversione in base a quanto indicato con gli specificatori di conversione. Altre funzioni equivalenti, con il nome che inizia con la lettera «v», hanno bisogno di un puntatore di tipo '**va_list**':

```
int vfscanf (FILE *restrict stream,
             const char *restrict conversione,
             va_list arg);
```

```
int vsscanf (const char *restrict s,
             const char *restrict conversione,
             va_list arg);
```

```
int vscanf  (const char *restrict conversione,
             va_list arg);
```

67.4 Riferimenti

«

- *Rationale for American National Standard for Information Systems - Programming Language - C: Input/Output*, http://www.lysator.liu.se/c/rat/d9.html
- *ISO/IEC 9899:TC2*, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf

[1] Nella tabella, in questa fase, non si distingue ancora tra accessi a file di testo rispetto a quelli relativi a file binari, pertanto non appare mai la sigla '**b**'.

[2] Può trattarsi anche di sequenze multibyte, ovvero di rappresentazioni dei caratteri che usano più byte per carattere.

[3] Questa è una semplificazione, perché ci sono altre funzioni dello stesso gruppo, che iniziano con la lettera '**v**', le quali alla fine hanno un puntatore di tipo '**va_list**'.

[4] Questa è una semplificazione, perché ci sono altre funzioni dello stesso gruppo, che iniziano con la lettera '**v**', le quali alla fine hanno un puntatore di tipo '**va_list**'.