Perl è un linguaggio di programmazione ***interpretato*** (o quasi) che quindi viene eseguito senza bisogno di generare un eseguibile binario. In questo senso, i programmi Perl sono degli script eseguiti dal programma '**`perl`**' che per convenzione dovrebbe essere collocato in '`/usr/bin/`'.

Perl è molto importante in tutti gli ambienti Unix e per questo è utile conoscerne almeno i rudimenti. Volendo fare una scala di importanza, subito dopo la programmazione con la shell Unix standard, viene la programmazione in Perl.

Alcuni degli esempi proposti nel capitolo, riportano dei riferimenti esterni a servizi *pastebin*, nei quali è possibile produrre una propria modifica del codice e vederne l'esito, senza bisogno di disporre di un elaboratore completo, munito di interprete Perl.

24.1 Struttura fondamentale

Dal momento che i programmi Perl vengono realizzati in forma di script, per convenzione occorre indicare il nome del programma interprete nella prima riga.

```
#!/usr/bin/perl
```

Per l'esecuzione di script da parte di un interprete non si può fare affidamento sul percorso di ricerca degli eseguibili (la variabile di ambiente ***PATH***), è quindi importante che il binario '**`perl`**' si trovi dove previsto. Questa posizione ('`/usr/bin/perl`') è quella standard ed è opportuno che sia rispettata tale consuetudine, altrimenti i programmi in Perl di altri autori non potrebbero funzionare nel proprio sistema senza una variazione di tutti i sorgenti.

Il buon amministratore di sistema farebbe bene a collocare dei collegamenti simbolici in tutte le posizioni in cui sarebbe possibile che venisse cercato l'eseguibile '**`perl`**': '`/bin/perl`', '`/usr/bin/perl`' e '`/usr/local/bin/perl`'.

Come si può intuire, il simbolo '`#`' rappresenta l'inizio di un commento.

```
#!/usr/bin/perl
#
# Esempio di intestazione e di commenti in Perl.
…
```

Un'altra convenzione che riguarda gli script Perl è l'estensione: '`.pl`', anche se l'utilizzo o meno di questa non costituisce un problema.

24.1.1 Istruzioni

Le istruzioni seguono la convenzione del linguaggio C, per cui terminano con un punto e virgola ('`;`') e i raggruppamenti di queste, detti anche blocchi, si fanno utilizzando le parentesi graffe ('**`{ }`**').

istruzione **;**

{ ***istruzione*** **;** ***istruzione*** **;** ***istruzione*** **;** **}**

Generalmente, un'istruzione può essere interrotta e ripresa nella riga successiva, dal momento che la sua conclusione è dichiarata chiaramente dal punto e virgola finale.

24.1.2 Nomi

I nomi che servono per identificare ciò che si utilizza all'interno del programma seguono regole determinate. In particolare:

- un nome può iniziare con un trattino basso o una lettera e può continuare con lettere, numeri e trattini bassi;
- i nomi sono sensibili alla differenza tra lettere maiuscole e minuscole.

Spesso i nomi sono preceduti da un simbolo che ne definisce il contesto:

Simbolo	Descrizione
`$`	il dollaro precede i nomi delle variabili scalari e degli elementi scalari di un array;
`@`	il simbolo *at* precede i nomi degli array normali o di raggruppamenti di elementi in essi contenuti;
`%`	il simbolo di percentuale precede i nomi degli array associativi, detti anche hash;
`&`	il simbolo e-commerciale precede i nomi delle funzioni quando queste vengono chiamate.

24.1.3 Contesto operativo

Perl è un linguaggio di programmazione con cui gli elementi che si indicano hanno un valore riferito al contesto in cui ci si trova. Questo significa, per esempio, che un array può essere visto come: una lista di elementi, il numero degli elementi contenuti, o una stringa contenente tutti i valori degli elementi contenuti.

In pratica, ciò serve a garantire che i dati siano trasformati nel modo più adatto al contesto, al quale è importante fare attenzione.

24.1.4 Tipi di dati

I tipi di dati più importanti che si possono gestire con Perl sono:

- stringhe;
- valori numerici;
- riferimenti;
- liste.

Le variabili di Perl vengono create semplicemente con l'assegnamento di un valore, senza la necessità di dichiarare il tipo o la dimensione. Le conversioni dei valori numerici sono fatte automaticamente in base al contesto.

In Perl non esiste un tipo di dati logico (nel senso di *Vero* o *Falso*); solo il risultato di una condizione lo è, ma non equivale a un valore gestibile in una variabile. Da un punto di vista logico-booleano, i valori seguenti vengono considerati equivalenti a *Falso*:

Valore	Descrizione
indefinito	equivalente a una variabile non dichiarata;
`""`	la stringa nulla;
`0`	il valore numerico zero;
`"0"`	la stringa corrispondente al numero zero.

Qualunque altro valore viene trattato come equivalente a *Vero*.

24.1.5 Esecuzione dei programmi Perl

Per poter eseguire un programma Perl, così come accade per qualunque altro tipo di script, occorre attivare il permesso di esecuzione per il file che lo contiene.

```
chmod +x programma_perl
```

Sembra banale o evidente, ma spesso ci si dimentica di farlo e quello che si ottiene è il classico ***permesso negato***: *Permission denied*.

24.2 Variabili e costanti scalari

La gestione delle variabili e delle costanti in Perl è molto simile a quella delle shell comuni. Una variabile scalare è quella che contiene un valore unico, contrapponendosi generalmente all'array che in Perl viene definito come variabile contenente una lista di valori.

24.2.1 Variabili

Le variabili scalari di Perl possono essere dichiarate in qualunque punto del programma e la loro dichiarazione coincide con l'inizializzazione, cioè l'assegnamento di un valore. I nomi delle variabili scalari iniziano sempre con il simbolo dollaro ('**$**').

L'utilizzo del dollaro come prefisso dei nomi delle variabili assomiglia a quanto si fa con le shell derivate da quella di Bourne, con la differenza che con Perl il dollaro si lascia sempre, mentre con queste shell si utilizza solo quando si deve leggere il loro contenuto.

```
$variabile_scalare = valore
```


L'assegnamento di un valore a una variabile scalare implica l'utilizzo di quanto si trova alla destra del simbolo di assegnamento ('**=**') come valore scalare: una stringa, un numero o un riferimento. È il contesto a decidere il risultato dell'assegnamento.

24.2.2 Variabili predefinite

Perl fornisce automaticamente alcune variabili scalari che normalmente non devono essere modificate dai programmi. Tali variabili servono per comunicare al programma alcune informazioni legate al sistema, oppure l'esito dell'esecuzione di una funzione, esattamente come accade con i parametri delle shell comuni. La tabella 24.5 mostra un elenco di alcune di queste variabili standard. Si può osservare che i nomi di tali variabili non seguono la regola per cui il primo carattere deve essere un trattino basso o una lettera. Questa eccezione consente di evitare di utilizzare inavvertitamente nomi corrispondenti a variabili predefinite.

Tabella 24.5. Elenco di alcune variabili standard di Perl.

Nome	Descrizione
`$$`	Numero PID del programma.
`$<`	Numero UID reale dell'utente che esegue il programma.
`$>`	Numero UID efficace dell'utente che esegue il programma.
`$?`	Lo stato dell'ultima chiamata di sistema.
`$_`	Argomento predefinito di molte funzioni.
`$0`	Il nome del programma.
`$"`	Separatore di lista.
`$/`	Separatore di righe per l'input (*input record separator*).

24.2.3 Costanti

Le costanti scalari più importanti sono di tipo stringa o numeriche. Le prime richiedono la delimitazione con apici doppi o singoli, mentre quelle numeriche non richiedono alcuna delimitazione.

Perl gestisce le stringhe racchiuse tra apici doppi in maniera simile a quanto fanno le shell tradizionali:

- le variabili indicate al loro interno vengono espanse, o meglio, ***interpolate*** (secondo la terminologia di Perl);
- la barra obliqua inversa ('\') può essere utilizzata come prefisso di escape quando si vogliono includere nella stringa simboli che altrimenti sarebbero interpretati in modo diverso e quando si vogliono indicare codici per cui non esiste un simbolo della tastiera.

Se una stringa viene interrotta e ripresa nella riga successiva, quello che si ottiene, nel punto dell'interruzione, è l'inserimento di un codice di interruzione di riga. In pratica, lo stesso codice di interruzione di riga utilizzato per andare a capo, viene inserito nella stringa e trattato esattamente per quello che è.

Anche le stringhe racchiuse tra apici singoli sono gestite in modo simile alle shell tradizionali:

- al loro interno non vengono effettuate interpolazioni di variabili;
- il carattere di escape, rappresentato dalla barra obliqua inversa, può essere utilizzato solo per inserire un apice letterale e la barra obliqua inversa stessa ('\'' e '\\').

Inoltre, davanti all'apice di inizio di una tale stringa, è necessario sia presente uno spazio.

La tabella 24.6 mostra un elenco di alcune di queste sequenze di escape utilizzabili nelle stringhe.

Tabella 24.6. Elenco di alcune sequenze di escape utilizzabili nelle stringhe delimitate con gli apici doppi.

Escape	Corrispondenza
`\\`	\
`\"`	"
`\$`	$
`\@`	@
`\'`	'
`\t`	<*HT*>
`\n`	<*LF*>
`\r`	<*CR*>
`\f`	<*FF*>
`\b`	<*BS*>
`\a`	<*BELL*>
`\e`	<*ESC*>
`\0`***n***	Numero ottale rappresentato da ***n***.
`\x`***h***	Numero esadecimale rappresentato da ***h***.

Quando all'interno di stringhe tra apici doppi si indicano delle variabili (scalari e non), potrebbe porsi un problema di ambiguità causato dalla necessità di distinguere il nome delle variabili dal resto della stringa. Quando dopo il nome della variabile segue un carattere o un simbolo che non può fare parte del nome (come uno spazio o un simbolo di punteggiatura), Perl non ha difficoltà a individuare la fine del nome della variabile e la continuazione della stringa. Quando ciò non è sufficiente, si può delimitare il nome della variabile tra parentesi graffe, così come si fa con le shell tradizionali.

```
${variabile}
```

```
@{variabile}
```

24.2.3.1 Costanti numeriche

Le costanti numeriche possono essere indicate nel modo consueto, quando si usa la numerazione a base decimale, oppure anche in esadecimale e in ottale.

Con la numerazione a base 10, si possono indicare interi nel modo normale e valori decimali utilizzando il punto come separazione tra la parte intera e la parte decimale. Si può utilizzare anche la notazione esponenziale.

- numero intero: '**123456**'
- numero intero leggibile più facilmente: '**1_234_567**'
- numero reale: '**123456.789**'
- notazione esponenziale: '**2.3E-10**'

Un numero viene trattato come esadecimale quando è preceduto dal prefisso '**0x**' e come ottale quando inizia con uno zero.

- numero esadecimale: '**0xFFFF**'
- numero ottale: '**0377**'

Quando un numero ottale o esadecimale è contenuto in una stringa, l'eventuale conversione in numero non avviene automaticamente, come invece accade in presenza di notazioni in base dieci.

24.2.4 Esempi

L'esempio seguente è il più banale, emette semplicemente la stringa '**"Ciao Mondo!\n"**' attraverso lo standard output. È da osservare la parte finale, '**\n**', che completa la stringa con un codice di interruzione di riga in modo da portare a capo il cursore in una nuova riga dello schermo.

Listato 24.7. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/KtcK8fRY* , *http://ideone.com/e2DWW* .

```
#!/usr/bin/perl

print "Ciao Mondo!\n";
```

Se il file si chiama '**1.pl**', lo si deve rendere eseguibile e quindi si può provare il suo funzionamento:

```
$ chmod +x 1.pl[Invio]

$ 1.pl[Invio]

  Ciao Mondo!
```

L'esempio seguente genera lo stesso risultato di quello precedente, ma con l'uso di variabili. Si può osservare che solo alla fine viene emesso il codice di interruzione di riga.

Listato 24.9. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/Jbvvdslx* , *http://ideone.com/2ZnJE* .

```
#!/usr/bin/perl

$primo = "Ciao";
$secondo = "Mondo";
print $primo;
print " ";
print $secondo;
print "!\n";
```

L'esempio seguente genera lo stesso risultato di quello precedente, ma con l'uso dell'interpolazione delle variabili all'interno di stringhe racchiuse tra apici doppi.

Listato 24.10. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/jkotRz4T* , *http://ideone.com/oRxzp* .

```
#!/usr/bin/perl

$primo = "Ciao";
$secondo = "Mondo";
print "$primo $secondo!\n";
```

L'esempio seguente emette la parola '**CiaoMondo**' senza spazi intermedi utilizzando la tecnica delle parentesi graffe.

Listato 24.11. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/3Gps1HJ3* , *http://ideone.com/o9TQx* .

```
#!/usr/bin/perl

$primo = "Ciao";
print "${primo}Mondo!\n";
```

L'esempio seguente mostra il comportamento degli apici singoli per delimitare le stringhe. Non si ottiene più l'interpolazione delle variabili.

Listato 24.12. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/0jGQRRic* , *http://ideone.com/cVdF7* .

```
#!/usr/bin/perl

$primo = "Ciao";
$secondo = "Mondo";
print '$primo $secondo!\n';
```

Se il file si chiama '**5.pl**', si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 5.pl[Invio]

$ 5.pl[Invio]

  $primo $secondo!\n
```

Inoltre, mancando il codice di interruzione di riga finale, l'invito della shell riappare subito alla destra di quanto visualizzato.

L'esempio seguente mostra l'uso di una costante e di una variabile numerica. Il valore numerico viene convertito automaticamente in stringa al momento dell'interpolazione.

Listato 24.14. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/AqwTcaA2* , *http://ideone.com/8TpI5* .

```
#!/usr/bin/perl

$volte = 1000;
$primo = "Ciao";
$secondo = "Mondo";
print "$volte volte $primo $secondo!\n";
```

Se il file si chiama '**`6.pl`**', si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 6.pl [Invio]

$ 6.pl [Invio]

 1000 volte Ciao Mondo!
```

L'esempio seguente permette di prendere confidenza con le variabili predefinite descritte in precedenza.

Listato 24.16. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/rr18JM1879* , *http://ideone.com/FfUIx* .

```
#!/usr/bin/perl

print "Nome del programma: $0\n";
print "PID del programma: $$\n";
print "UID dell'utente: $<\n";
print "Ultima chiamata di sistema: $?\n";
```

Se il file si chiama '**`7.pl`**', si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 7.pl [Invio]

$ 7.pl [Invio]
```

Il risultato potrebbe essere simile a quello seguente:

```
Nome del programma: ./7.pl
PID del programma: 717
UID dell'utente: 500
Ultima chiamata di sistema: 0
```

24.3 Array e liste

Perl gestisce gli array in modo dinamico, nel senso che possono essere allungati e accorciati a piacimento. Quando si parla di array si pensa generalmente a una variabile che abbia questa forma; ma Perl permette di gestire delle costanti array, definite liste.

Generalmente, il primo elemento di un array o di una lista ha indice zero. Questo assunto può essere cambiato agendo su una particolare variabile predefinita, ma ciò è sconsigliabile.

24.3.1 Liste

Le liste sono una sequenza di elementi scalari, di qualunque tipo, separati da virgole, racchiusi tra parentesi tonde. L'ultimo elemento può essere seguito o meno da una virgola, prima della parentesi chiusa.

```
(elemento, ...)
```

La lista vuota, o nulla, si rappresenta con le semplici parentesi aperta e chiusa:

```
()
```

Seguono alcuni esempi in cui si mostrano diversi modi di indicare la stessa lista.

```
("uno", "due", "tre", "quattro", "ciao")
```



```
("uno", "due", "tre", "quattro", "ciao",)
```

```
("uno",
 "due",
 "tre",
 "quattro",
 "ciao",)
```

```
(
    "uno",
    "due",
    "tre",
    "quattro",
    "ciao",
)
```

Una lista può essere utilizzata per inizializzare un array, ma se si pretende di assegnare una lista a un variabile scalare, si ottiene in pratica che la variabile scalare contenga solo il valore dell'ultimo elemento della lista (alla variabile vengono assegnati, in sequenza, tutti gli elementi della lista, per cui, quello che resta è l'ultimo). Per esempio:

```
$miavar = ("uno", "due", "tre", "quattro", "ciao");
```

assegna a '**`$miavar`**' solo la stringa '**`"ciao"`**'.

Una lista di valori può essere utilizzata con un indice, per fare riferimento solo a uno di tali valori. Naturalmente ciò è utile quando l'indice è rappresentato da una variabile. L'esempio seguente mostra la trasformazione di un indice ('**`$ind`**'), che abbia un valore numerico compreso tra zero e nove, in un termine verbale.

```
$numverb = (
        "zero",
        "uno",
        "due",
        "tre",
        "quattro",
        "cinque",
        "sei",
        "sette",
        "otto",
        "nove",
)[$ind];
```

Gli elementi contenuti in una lista che non sono scalari, vengono interpolati, incorporando in quel punto tutti gli elementi che questi rappresentano. Gli eventuali elementi non scalari nulli, non rappresentano alcun elemento e vengono semplicemente ignorati. Per esempio, le due liste seguenti sono perfettamente identiche:

```
("uno", "due", (), ("tre", "quattro", "cinque"), "sei")
```

```
("uno", "due", "tre", "quattro", "cinque", "sei")
```

Naturalmente ciò ha maggiore significato quando non si tratta semplicemente di liste annidate, ma di array collocati all'interno di liste.

24.3.2 Array

L'array è una variabile contenente una lista di valori di qualunque tipo, purché scalari. Il nome di un array inizia con il simbolo '**`@`**' quando si fa riferimento a tutto l'insieme dei suoi elementi o anche solo a parte di questi. Quando ci si riferisce a un solo elemento di questo si utilizza il dollaro.

In pratica, quando si fa riferimento a un solo elemento di un array si può immaginare che si tratti di un gruppo di elementi composto da un solo elemento, per cui si può utilizzare il prefisso '**`@`**' anche in questo caso.

Un array può essere dichiarato vuoto, con la sintassi seguente:

```
@array = ()
```

In alternativa gli si può assegnare una lista di elementi:

```
@array = (elemento, ...)
```

Il riferimento a un solo elemento di un array viene indicato con la notazione seguente (le parentesi quadre fanno parte della notazione):

```
$array[indice]
```

Il riferimento a un raggruppamento di elementi può essere indicato in vari modi:

```
@array[indice1, indice2, ...]
```

In tal caso ci si riferisce a un sottoinsieme composto dagli elementi indicati dagli indici contenuti all'interno delle parentesi quadre.

```
@array[indice_iniziale . . indice_finale]
```

In questo modo ci si riferisce a un sottoinsieme composto dagli elementi contenuti nell'intervallo espresso dagli indici iniziale e finale.

Nella gestione degli array sono importanti due variabili predefinite:

Variabile	Descrizione
`$[`	rappresenta l'indice del primo elemento di un array e si usa azzerata convenzionalmente, in modo che per identificare il primo elemento serva l'indice zero (meglio non modificare questa variabile);
`$#`*array*	rappresenta l'ultimo indice dell'array identificato dal nome posto dopo il simbolo '**$#**'.

Assegnare un array o parte di esso a una variabile scalare, significa in pratica assegnare un numero intero esprimente il numero di elementi in esso contenuti. L'esempio seguente assegna in pratica a '**$mioscalare**' il valore due.

```
@mioarray = ("uno", "due");
$mioscalare = @mioarray;
```

Inserire un array o parte di esso in una stringa delimitata con gli apici doppi, implica l'interpolazione degli elementi, separati con quanto contenuto nella variabile '**$"**' (il separatore di lista). La variabile predefinita '**$"**' contiene normalmente uno spazio singolo. L'esempio seguente assegna a '**$mioscalare**' la stringa '**"uno due"**':

```
@mioarray = ("uno", "due");
$mioscalare = "@mioarray";
```

Perl fornisce degli array predefiniti, di cui il più importante è '**@ARGV**' che contiene l'elenco degli argomenti ricevuti dalla riga di comando.

L'esempio seguente permette di verificare quanto descritto sugli array di Perl.

Listato 24.29. Per provare il codice attraverso un servizio *pastebin*: http://codepad.org/5TgVcUXc , http://ideone.com/qL3q6 .

```
#!/usr/bin/perl

# Dichiara l'array assegnandogli sia stringhe che numeri
@elenco = ("primo", "secondo", 3, 4, "quinto");

# Attraverso l'assegnamento seguente, $elementi riceve il
# numero di elementi contenuti nell'array.
$elementi = @elenco;

# Emette tutte le informazioni legate all'array.
print "L'array contiene $elementi elementi.\n";
print "L'indice iniziale è $[.\n";
print "L'ultimo elemento si raggiunge con l'indice ";
print "$#elenco.\n";

# Emette in ordine tutti gli elementi dell'array.
print "L'array contiene: $elenco[0] $elenco[1] $elenco[2] ";
print "$elenco[3] $elenco[4].\n";

# Idem
print "Anche in questo modo si legge il contenuto ";
print "dell'array: @elenco.\n";
```

Se il file si chiama '**11.pl**', si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 11.pl [Invio]

$ 11.pl [Invio]

  L'array contiene 5 elementi.
  L'indice iniziale è 0.
  L'ultimo elemento si raggiunge con l'indice 4.
  L'array contiene: primo secondo 3 4 quinto.
  Anche in questo modo si legge il contenuto dell'array: ↩
↪primo secondo 3 4 quinto.
```

L'esempio seguente mostra il funzionamento dell'array predefinito '**@ARGV**':

```
#!/usr/bin/perl

print "Il programma $0 è stato avviato con gli argomenti\n";
print "seguenti:\n";
print "@ARGV\n";
print "Il primo argomento era $ARGV[0]\n";
print "e l'ultimo era $ARGV[$#ARGV].\n";
```

Se il file si chiama '**12.pl**', si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 12.pl [Invio]

$ 12.pl carbonio idrogeno ossigeno [Invio]

  Il programma ./12.pl è stato avviato con gli argomenti
  seguenti:
  carbonio idrogeno ossigeno
  Il primo argomento era carbonio
  e l'ultimo era ossigeno.
```

24.4 Array associativi o hash

L'array associativo, o hash, è un tipo speciale di array che normalmente non si trova negli altri linguaggi di programmazione. Gli elementi sono inseriti a coppie, dove il primo elemento della coppia è la chiave di accesso per il secondo.

Il nome di un hash inizia con il segno di percentuale ('**%**'), mentre il riferimento a un elemento scalare di questo si fa utilizzando il dollaro, mentre l'indicazione di un sottoinsieme avviene con il simbolo '**@**', come per gli array.

La dichiarazione, ovvero l'assegnamento di un array associativo, si esegue in uno dei due modi seguenti:

```
%array_associativo = (chiave, elemento, ...)
```

```
%array_associativo = (chiave => elemento, ...)
```

La seconda notazione esprime meglio la dipendenza tra la chiave e l'elemento che con essa viene raggiunto. L'elemento che funge da chiave viene trattato sempre come stringa, mentre gli elementi abbinati alle chiavi possono essere di qualunque tipo scalare. In particolare, nel caso si utilizzi l'abbinamento tra chiave e valore attraverso il simbolo '**=>**', ciò che sta alla sinistra di questo viene interpretato come stringa in ogni caso, permettendo di eliminare la normale delimitazione attraverso apici.

Un elemento singolo di un hash viene indicato con la notazione seguente, dove le parentesi graffe fanno parte dell'istruzione.

```
$array_associativo{chiave}
```

La chiave può essere una costante stringa o un'espressione che restituisce una stringa. La costante stringa può anche essere indicata senza apici.

Un sottoinsieme di un hash è un'entità equivalente a un array e viene indicato con la notazione seguente:

```
@array_associativo{chiave1, chiave2, ...}
```

Perl fornisce alcuni array associativi predefiniti. Il più importante è '**%ENV**' che contiene le variabili di ambiente, cui si accede indicando il nome della variabile come chiave.

L'esempio seguente mostra un semplice array associativo e il modo di accedere ai suoi elementi in base alla chiave:

Listato 24.33. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/j6SWnMby* , *http://ideone.com/jH8I9* .

```
#!/usr/bin/perl

# Dichiarazione dell'array: attenzione a non fare confusione!
#            -------------    ----------------    ---------
%deposito = ("primo", "alfa", "secondo", "bravo", "terzo", 3);


# Emette il contenuto dei vari elementi.
print "$deposito{primo}\n";
print "$deposito{secondo}\n";
print "$deposito{terzo}\n";
```

Se il file si chiama '**21.pl**', si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 21.pl [Invio]

$ 21.pl [Invio]

alfa
bravo
3
```

L'esempio seguente è identico al precedente, ma l'hash viene dichiarato in modo più facile da interpretare visivamente.

Listato 24.35. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/V0L3pAhu* , *http://ideone.com/xHqM2* .

```
#!/usr/bin/perl

# Dichiarazione dell'array.
%deposito = (
        "primo", "alfa",
        "secondo", "bravo",
        "terzo", 3,
);

# Emette il contenuto dei vari elementi.
print "$deposito{primo}\n";
print "$deposito{secondo}\n";
print "$deposito{terzo}\n";
```


L'esempio seguente è identico al precedente, ma l'hash viene dichiarato in modo ancora più leggibile.

Listato 24.36. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/4m0QjyCW* , *http://ideone.com/jnRPE* .

```
#!/usr/bin/perl

# Dichiarazione dell'array.
%deposito = (
        primo   => "alfa",
        secondo => "bravo",
        terzo   => 3,
);

# Emette il contenuto dei vari elementi.
print "$deposito{primo}\n";
print "$deposito{secondo}\n";
print "$deposito{terzo}\n";
```

L'esempio seguente mostra l'uso dell'array '**%ENV**' per la lettura delle variabili di ambiente:

Listato 24.37. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/8CfLW2hc* , *http://ideone.com/RkSG8* .

```
#!/usr/bin/perl

print "PATH: $ENV{PATH}\n";
print "TERM: $ENV{TERM}\n";
```

Se il file si chiama '**24.pl**', si può verificare il suo funzionamento nel modo seguente:

```
$ chmod +x 24.pl [Invio]

$ 24.pl [Invio]

PATH: /usr/local/bin:/bin:/usr/bin:/usr/bin/X11
TERM: linux
```

24.5 Operatori ed espressioni

Il sistema di operatori e delle relative espressioni che possono essere create con Perl è piuttosto complesso. La parte più consistente di questa gestione riguarda il trattamento delle stringhe, che qui viene descritto particolarmente in un altro capitolo. Alcuni tipi di espressioni e i relativi operatori non vengono mostrati, data la loro complessità per chi non conosca già il linguaggio C. In particolare viene saltata la gestione dei dati a livello di singoli bit.

Il senso e il risultato di un'espressione dipende dal contesto. La valutazione di un'espressione dipende dalle precedenze che esistono tra i vari tipi di operatori. Si parla di precedenza superiore quando qualcosa viene valutato prima di qualcos'altro, mentre la precedenza è inferiore quando qualcosa viene valutato dopo qualcos'altro.

24.5.1 Operatori che intervengono su valori numerici, stringhe e liste

Gli operatori che intervengono su valori numerici sono elencati nella tabella 24.39.

Tabella 24.39. Elenco degli operatori utilizzabili in presenza di valori numerici. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
++op	Incrementa di un'unità l'operando prima che venga restituito il suo valore.
op++	Incrementa di un'unità l'operando dopo averne restituito il suo valore.
--op	Decrementa di un'unità l'operando prima che venga restituito il suo valore.
op--	Decrementa di un'unità l'operando dopo averne restituito il suo valore.
+op	Non ha alcun effetto.
-op	Inverte il segno dell'operando.

Operatore e operandi	Descrizione
op1 + *op2*	Somma i due operandi.
op1 - *op2*	Sottrae dal primo il secondo operando.
op1 * *op2*	Moltiplica i due operandi.
op1 / *op2*	Divide il primo operando per il secondo.
op1 % *op2*	Modulo: il resto della divisione tra il primo e il secondo operando.
op1 ** *op2*	Eleva il primo operando alla potenza del secondo.
var = *valore*	Assegna alla variabile il valore alla destra.
op1 += *op2*	*op1* = *op1* + *op2*
op1 -= *op2*	*op1* = *op1* - *op2*
op1 *= *op2*	*op1* = *op1* * *op2*
op1 /= *op2*	*op1* = *op1* / *op2*
op1 %= *op2*	*op1* = *op1* % *op2*
op1 **= *op2*	*op1* = *op1* ** *op2*
op1 == *op2*	*Vero* se gli operandi si equivalgono.
op1 != *op2*	*Vero* se gli operandi sono differenti.
op1 < *op2*	*Vero* se il primo operando è minore del secondo.
op1 > *op2*	*Vero* se il primo operando è maggiore del secondo.
op1 <= *op2*	*Vero* se il primo operando è minore o uguale al secondo.
op1 >= *op2*	*Vero* se il primo operando è maggiore o uguale al secondo.

La gestione da parte di Perl delle stringhe è molto sofisticata e questa si attua principalmente attraverso gli operatori di delimitazione. In questa sezione si vuole solo accennare agli operatori che hanno effetto sulle stringhe, sorvolando su raffinatezze che si possono ottenere in casi particolari. La tabella 24.40 elenca tali operatori.

Tabella 24.40. Elenco degli operatori utilizzabili in presenza di valori alfanumerici, o stringa. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
str1 . *str2*	Concatena le due stringhe.
str x *num*	Restituisce la stringa ripetuta consecutivamente ***num*** volte.
str =~ *modello*	Collega il modello alla stringa. Il risultato dipende dal contesto.
str !~ *modello*	Come '**=~**', ma restituisce un valore inverso.
var = *valore*	Assegna alla variabile il valore alla destra.
op1 x= *op2*	*op1* = *op1* x *op2*
op1 .= *op2*	*op1* = *op1* . *op2*
str1 eq *str2*	*Vero* se le due stringhe sono uguali.
str1 ne *str2*	*Vero* se le due stringhe sono differenti.
str1 lt *str2*	*Vero* se la prima stringa è lessicograficamente inferiore alla seconda.
str1 gt *str2*	*Vero* se la prima stringa è lessicograficamente superiore alla seconda.
str1 le *str2*	*Vero* se la prima stringa è lessicograficamente inferiore o uguale alla seconda.
str1 ge *str2*	*Vero* se la prima stringa è lessicograficamente superiore o uguale alla seconda.

Gli operatori che intervengono sulle liste sono elencati nella tabella 24.41.

Tabella 24.41. Elenco degli operatori utilizzabili in presenza di liste. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
lista x *num*	Restituisce la lista composta ripetendo quella indicata per ***num*** volte.
array = *lista*	Crea l'array assegnandogli la lista indicata alla destra.
elem1 , *elem2*	La virgola è l'operatore di separazione degli elementi di una lista.
elem1 => *elem2*	Sinonimo della virgola.
elem1 .. *elem2*	Rappresenta una lista di valori da ***elem1*** a ***elem2***.

24.5.2 Operatori logici

È il caso di ricordare che con Perl tutti i tipi di dati possono essere valutati in modo logico: lo zero numerico o letterale, la stringa nulla e un valore indefinito corrispondono a *Falso*, in tutti gli altri casi si considera equivalente a *Vero*. Gli operatori logici sono elencati nella tabella 24.42.

Tabella 24.42. Elenco degli operatori logici. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
! *op*	Inverte il risultato logico dell'operando.
op1 && *op2*	Se il risultato del primo operando è *Falso* non valuta il secondo.
op1 || *op2*	Se il risultato del primo operando è *Vero* non valuta il secondo.
op1 and *op2*	Come '**&&**', ma con un livello di precedenza molto basso.
op1 or *op2*	Come '**||**', ma con un livello di precedenza molto basso.

Il risultato di un'espressione logica complessa è quello dell'ultima espressione elementare a essere valutata. Questo particolare è importante, anche se si tratta di un comportamento comune di diversi linguaggi, perché viene usato spesso per condizionare l'esecuzione di istruzioni, senza usare le strutture tradizionali, come *if-else*, o simili.

Questo tipo di approccio da parte del programmatore è sconsigliabile in generale, dato che serve a complicare la lettura e l'interpretazione umana del sorgente; tuttavia è importante conoscere esempi di questo tipo, perché sono sempre molti i programmi fatti alla svelta senza pensare alla leggibilità.

L'esempio seguente dovrebbe dare l'idea di come si può utilizzare l'operatore logico '**||**' (OR). Il risultato logico finale non viene preso in considerazione, quello che conta è solo il risultato della condizione '**$valore > 90**', che se non si avvera fa sì che venga eseguita l'istruzione '**print**' posta come secondo operando.

```
#!/usr/bin/perl
…
$valore = 100;
…
$valore > 90 || print "Il valore è insufficiente\n";
…
```

In pratica, se il valore contenuto nella variabile '**$valore**' supera 90, non si ottiene l'emissione del messaggio attraverso lo standard output. In questi casi, si usano preferibilmente gli operatori '**and**' e '**or**', che si distinguono perché hanno una precedenza molto bassa, adattandosi meglio alla circostanza.

```
$valore > 90 or print "Il valore è insufficiente\n";
```

Come si vede dalla variante dell'esempio proposta, l'espressione diventa quasi simpatica, perché sembra una frase inglese più comprensibile. La cosa può diventare ancora più «divertente» se si utilizza la funzione interna ***die()***, che serve a visualizzare un messaggio attraverso lo standard error e a concludere il funzionamento del programma Perl.

```
$valore > 90 or die "Il valore è insufficiente\n";
```

A parte la simpatia o il divertimento nello scrivere codice del genere, è bene ricordare che poi si tratta di qualcosa che un altro programmatore può trovare difficile da interpretare.

24.5.3 Operatori particolari

Tra gli operatori che non sono stati indicati nelle categorie descritte precedentemente, il più interessante è il seguente:

```
condizione ? espressione1 : espressione2
```

Se la condizione restituisce il valore *Vero*, allora l'operatore restituisce il valore della prima espressione, altrimenti quello della seconda.

24.5.4 Raggruppamenti di espressioni

Le espressioni, di qualunque genere siano, possono essere raggruppate in modo che la loro valutazione avvenga in un ordine differente da quanto previsto dalle precedenze legate agli operatori utilizzati. Per questo si usano le parentesi tonde, come avviene di solito anche negli altri linguaggi.

Le parentesi tonde sono anche i delimitatori delle liste, per cui è anche possibile immaginare che esistano delle liste contenenti delle espressioni. Se si valuta una lista di espressioni, si ottiene il risultato della valutazione dell'ultima di queste.

24.6 Strutture di controllo del flusso

Perl gestisce praticamente tutte le strutture di controllo di flusso degli altri linguaggi di programmazione, compreso *go-to* che comunque è sempre meglio non utilizzare e qui non viene presentato volutamente.

Quando una struttura particolare controlla un gruppo di istruzioni, queste vengono delimitate necessariamente attraverso le parentesi graffe, come avviene in C, ma a differenza di quel linguaggio non è possibile farne a meno quando ci si limita a indicare una sola istruzione.

Le strutture di controllo del flusso basano normalmente questo controllo sulla verifica di una condizione espressa all'interno di parentesi tonde.

Nei modelli sintattici indicati, le parentesi graffe fanno parte delle istruzioni, essendo i delimitatori dei blocchi di istruzioni di Perl.

24.6.1 Strutture condizionali: «if» e «unless»

```
if (condizione) { istruzione;…}
```

```
if (condizione) { istruzione;…} else { istruzione;…}
```

```
if (cond) { istr;…} elsif (cond) { istr;…}… else { istr;…}
```

Se la condizione si verifica viene eseguito il gruppo di istruzioni seguente, racchiuso tra parentesi graffe, quindi il controllo passa alle istruzioni successive alla struttura. Se viene utilizzato '**elsif**', nel caso non si verifichino altre condizioni precedenti, viene verificata la condizione successiva; se questa si avvera, viene eseguito il gruppo di istruzioni che ne segue. Al termine il controllo riprende dopo la struttura. Se viene utilizzato '**else**', quando non si verifica alcuna condizione di quelle poste, viene eseguito il gruppo di istruzioni finale. Vengono mostrati alcuni esempi:

```
if ($importo > 10000000) { print "L'offerta è vantaggiosa"; }
```

```
if ($importo > 10000000)
  {
    $memorizza = $importo;
    print "L'offerta è vantaggiosa.\n";
  }
else
  {
    print "Lascia perdere.\n";
  }
```

```
if ($importo > 10000000)
  {
    $memorizza = $importo;
    print "L'offerta è vantaggiosa.\n";
  }
elsif ($importo > 5000000)
  {
    $memorizza = $importo;
    print "L'offerta è accettabile.\n";
  }
else
  {
    print "Lascia perdere.\n";
  }
```

La parola '**unless**' può essere utilizzata come '**if**', con la differenza che la condizione viene valutata in modo opposto, cioè viene eseguito il gruppo di istruzioni che segue '**unless**' solo se **non** si verifica la condizione.

24.6.2 Iterazioni con condizione di uscita iniziale: «while» e «until»

```
while (condizione) { istruzione;…}
```

```
while (condizione) { istruzione;…} continue { istruzione;…;}
```

La struttura '**while**' esegue un gruppo di istruzioni finché la condizione restituisce il valore *Vero*. La condizione viene valutata prima di eseguire il gruppo di istruzioni e poi ogni volta che termina un ciclo, prima dell'esecuzione del successivo.

Il blocco di istruzioni che segue '**continue**' viene eseguito semplicemente di seguito al gruppo normale. Ci sono situazioni in cui viene saltato. Segue l'esempio del calcolo del fattoriale.

Listato 24.49. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/PoCiw0e5* , *http://ideone.com/zqCNt* .

```
#!/usr/bin/perl

# Il numero di partenza viene fornito come argomento nella
# riga di comando.
$numero = $ARGV[0];
$cont = $numero -1;
while ($cont > 0)
  {
    $numero = $numero * $cont;
    $cont = $cont -1;
  }
print "Il fattoriale è $numero.\n";
```

La stessa cosa si potrebbe semplificare nel modo seguente.

Listato 24.50. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/6HKWQBEt* , *http://ideone.com/BiDjj* .

```
#!/usr/bin/perl
#
# Il numero di partenza viene fornito come argomento nella
# riga di comando.
#
$numero = $ARGV[0];
$cont = $numero -1;
while ($cont)
  {
    $numero *= $cont;
    $cont--;
  }
print "Il fattoriale è $numero.\n";
```

All'interno delle istruzioni di un ciclo '**while**' possono apparire alcune istruzioni particolari:

next	interrompe l'esecuzione del gruppo di istruzioni e riprende dalla valutazione della condizione (se esiste il gruppo '**continue**', '**next**' rinvia all'esecuzione di questo e quindi alla valutazione della condizione);
last	esce definitivamente dal ciclo '**while**' senza curarsi del gruppo di istruzioni '**continue**';
redo	ripete il ciclo, senza valutare e verificare nuovamente l'espressione della condizione e senza curarsi del gruppo di istruzioni '**continue**'.

L'esempio seguente è una variante del calcolo del fattoriale in modo da vedere il funzionamento di '**last**'. Si osservi che '**while (1){…}**' equivale a un ciclo senza fine perché la condizione (cioè il valore 1) è sempre vera.

Listato 24.52. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/N2aY4IBz* , *http://ideone.com/VW3Pq* .

```
#!/usr/bin/perl
#
# Il numero di partenza viene fornito come argomento
# nella riga di comando.
#
$numero = $ARGV[0];
$cont = $numero -1;
# Il ciclo seguente è senza fine.
while (1)
  {
    $numero *= $cont;
    $cont--;
    if (!$cont)
      {
        last;
      }
  }
print "Il fattoriale è $numero.\n";
```

La parola '**until**' può essere utilizzata al posto di '**while**', con la differenza che la condizione viene valutata in modo opposto, cioè viene eseguito il gruppo di istruzioni che segue '**until**' solo se **non** si verifica la condizione. In pratica, al verificarsi della condizione, il ciclo termina.

24.6.3 Iterazioni con condizione di uscita finale: «do-while» e «do-until»

```
do { istruzione;…} while (condizione)
```

La struttura '**do**…**while**' esegue un gruppo di istruzioni almeno una volta, quindi ne ripete l'esecuzione finché la condizione restituisce il valore *Vero*. Segue il solito esempio del calcolo del fattoriale:

Listato 24.53. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/4ANM1LUe* , *http://ideone.com/qHxi3* .

```
#!/usr/bin/perl
#
# Il numero di partenza viene fornito come argomento
# nella riga di comando.
#
$cont       = $ARGV[0];
$fattoriale = 1;
do
  {
    $fattoriale *= $cont;
    $cont--;
  }
while ($cont);
print "Il fattoriale è $fattoriale.\n";
```

L'uso della parola '**until**', al posto di '**while**', fa sì che la verifica della condizione avvenga nel senso che non si avveri, in pratica inverte il senso della condizione che controlla l'uscita dal ciclo.

24.6.4 Iterazione enumerativa: «for»

```
for (espressione1; espressione2; espressione3) { istruzione;…}
```

Questa è la forma tipica di un'istruzione '**for**', in cui la prima espressione corrisponde all'assegnamento iniziale di una variabile, la seconda a una condizione che deve verificarsi fino a che si vuole che sia eseguito il gruppo di istruzioni e la terza all'incremento o decremento della variabile inizializzata con la prima espressione. In pratica, potrebbe esprimersi nella sintassi seguente:

```
for ($var = n ; condizione; $var++) { istruzione;…}
```

In realtà la forma del ciclo '**for**' potrebbe essere diversa, ma in tal caso si preferisce utilizzare il nome '**foreach**' che è comunque un sinonimo.

In breve: la prima espressione viene eseguita una volta sola all'inizio del ciclo; la seconda viene valutata all'inizio di ogni ciclo e il gruppo di istruzioni viene eseguito solo se il risultato è *Vero*. L'ultima espressione viene eseguita alla fine dell'esecuzione del gruppo di istruzioni, prima che si ricominci con l'analisi della condizione.

Segue il solito esempio del calcolo del fattoriale:

Listato 24.54. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/ribOo6ds* , *http://ideone.com/uczTi* .

```
#!/usr/bin/perl
#
# Il numero di partenza viene fornito come argomento
# nella riga di comando.
#
$input = $ARGV[0];
$numero = $input;
for ($cont = 1; $cont < $input; $cont++)
  {
    $numero *= $cont;
  }
print "Il fattoriale è $numero.\n";
```

24.6.5 Iterazione con scansione di valori: «foreach»

```
foreach var_scalare lista { istruzione;…}
```

La parola '**foreach**' è un sinonimo di '**for**', per cui si tratta della stessa cosa, solo che si preferisce utilizzare due termini differenti per una struttura che può articolarsi in due modi alternativi.

La variabile scalare iniziale, viene posta di volta in volta ai valori contenuti nella lista, eseguendo ogni volta il gruppo di istruzioni. Il ciclo finisce quando non ci sono più elementi nella lista.

Segue il solito esempio del calcolo del fattoriale:

Listato 24.55. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/JhWPVfOH* , *http://ideone.com/jpYpL* .

```
#!/usr/bin/perl
#
# Il numero di partenza viene fornito come argomento
# nella riga di comando.
#
$input = $ARGV[0];
$numero = $input;
foreach $cont (1 .. ($input -1))
  {
    $numero *= $cont;
  }
print "Il fattoriale è $numero.\n";
```

24.6.6 Istruzioni condizionate

Una brutta tradizione di Perl consente la scrittura di istruzioni condizionate secondo le sintassi seguenti:

```
espressione1 if espressione2
```

```
espressione1 unless espressione2
```

```
espressione1 while espressione2
```

```
espressione1 until espressione2
```

Si tratta di forme abbreviate e sconsigliabili (secondo il parere di chi scrive) delle sintassi seguenti.

```
if (espressione2) { espressione1 }
```

```
unless (espressione2) { espressione1 }
```

```
while (espressione2) { espressione1 }
```

```
until (espressione2) { espressione1 }
```

Come si vede, lo sforzo necessario a scrivere le istruzioni nel modo normale, è minimo. Evidentemente, l'idea che sta alla base della possibilità di usare sintassi così strane delle strutture '**if**', '**while**' e simili, è quella di permettere la scrittura di codice che assomigli alla lingua inglese.

24.7 Funzioni interne

Perl fornisce una serie di funzioni già pronte. In realtà, più che di funzioni vere e proprie, si tratta di operatori unari che intervengono sull'argomento posto alla loro destra. Questa precisazione è importante perché serve a comprendere meglio il meccanismo con cui Perl interpreta le chiamate di tali funzioni od operatori.

Finora si è visto il funzionamento di una funzione molto semplice, '**print**'. Questa emette il risultato dell'operando che si trova alla sua destra, ma solo del primo. Se ciò che appare alla destra di '**print**' è un'espressione, la valutazione dell'insieme '**print** ***espressione***', dipende dalle precedenze tra gli operandi. Infatti:

-
```
print 1+2+4;
```
restituisce sette;

-
```
print (1+2)+4;
```
restituisce tre;

-
```
print (1+2+4);
```
restituisce sette.

Utilizzando le funzioni di Perl nello stesso modo in cui si fa negli altri linguaggi, racchiudendo l'argomento tra parentesi, si evitano ambiguità; soprattutto, in questo modo, sembrano essere veramente funzioni anche se si tratta di operatori.

L'argomento di queste funzioni di Perl (ovvero l'operando) può essere uno scalare o una lista. In questo caso quindi, così come lo scalare non ha la necessità di essere racchiuso tra parentesi, anche la lista non lo ha. Resta in ogni caso il fatto che ciò sia almeno consigliabile per migliorare la leggibilità del programma. La sezione 24.17 elenca e descrive alcune di queste funzioni.

24.8 Input e output dei dati

L'I/O può avvenire sia attraverso l'uso dei flussi standard di dati (standard input, standard output e standard error), sia utilizzando file differenti. I flussi di dati standard sono trattati come file normali, con la differenza che generalmente non devono essere aperti o chiusi.

Assieme alla gestione dei file si affianca la possibilità di eseguire comandi del sistema operativo, in parte descritta nella sezione dedicata agli operatori di delimitazione di stringhe.

24.8.1 Esecuzione di comandi di sistema

Una stringa racchiusa tra apici inversi, oppure indicata attraverso l'operatore di stringa '**qx**', viene interpolata e il risultato viene fatto eseguire dal sistema operativo.

L'output del comando è il risultato della valutazione della stringa e il valore restituito dal comando può essere letto dalla variabile predefinita '**$?**'. È importante ricordare che generalmente i comandi del sistema operativo restituiscono un valore pari a zero quando l'operazione ha avuto successo. Dal punto di vista di Perl, quando '**$?**' contiene il valore *Falso* significa che il comando ha avuto successo.

L'esempio seguente dovrebbe rendere l'idea:

```
#!/usr/bin/perl
#
# $elenco riceve l'elenco di file in forma di un'unica
# stringa.
#
$elenco = `ls *.pl`;

if ($? == 0)
  {
    #
    # L'operazione ha avuto successo e viene visualizzato
    # l'elenco.
    #
    print "$elenco\n";
  }
else
  {
    #
    # L'operazione è fallita.
    #
    print "Non ci sono programmi Perl\n";
  }
```

24.8.2 Gestione dei file

Perl, come molti altri linguaggi, gestisce i file aperti come flussi di file, o *file handle*. I flussi di file vengono indicati attraverso un nome che per convenzione è espresso quasi sempre attraverso lettere maiuscole.

Perl mette a disposizione tre flussi di file predefiniti: '**STDIN**', '**STDOUT**' e '**STDERR**'. Questi corrispondono rispettivamente ai flussi di standard input, standard output e standard error. Altri file possono essere utilizzati aprendoli attraverso la funzione ***open()***, con cui si abbina un flusso al file reale.

Perl è predisposto per gestire agevolmente i file di testo, cioè quelli organizzati convenzionalmente in righe terminanti con il codice di interruzione di riga. Si valuta un flusso di file come se si trattasse di una variabile, racchiudendone il nome tra parentesi angolari, ottenendo la lettura e la restituzione di una riga, ogni volta che avviene tale valutazione.

```
#!/usr/bin/perl

while (defined ($riga = <STDIN>))
 {
    print $riga;
 }
```

L'esempio appena mostrato emette attraverso lo standard output ciò che riceve dallo standard input. Quindi, la lettura del flusso di file attraverso la semplice valutazione dell'espressione, restituisce una riga fino al codice di interruzione di riga incluso. In questo modo, nell'esempio non è necessario aggiungere il codice '**\n**' nell'argomento della funzione '**print**'.

Se un flusso di file è l'unica cosa che appare nella condizione di un ciclo '**while**' o '**for**', la sua valutazione genera la lettura della riga e il suo inserimento all'interno della variabile predefinita '**$**_'. Questo fatto può essere usato convenientemente considerando che quando si raggiunge la fine, la valutazione del flusso di file genera un valore indefinito, pari a *Falso* in una condizione. I due esempi seguenti sono identici al quello mostrato poco sopra.

```
#!/usr/bin/perl

while (<STDIN>)
  {
    print $_;
  }
```

```
#!/usr/bin/perl

for ( ; <STDIN>; )
  {
    print $_;
  }
```

Un flusso di file può essere valutato in un contesto lista. In tal caso restituisce tutto il file in una lista in cui ogni elemento è una riga. Naturalmente ciò viene fatto a spese della memoria di elaborazione.

Listato 24.60. Per provare il codice attraverso un servizio *pastebin*: *http://ideone.com/apQjT* .

```
#!/usr/bin/perl

@mio_file = <STDIN>;
print @mio_file;
```

L'esempio appena mostrato si comporta come gli altri visti finora: restituisce lo standard input attraverso lo standard output. Si osservi che la funzione '**print**' ha l'argomento senza virgolette perché altrimenti inserirebbe uno spazio indesiderato tra un elemento e l'altro.

24.8.3 File globbing

Perl, se non riconosce ciò che trova all'interno di parentesi angolari come un flusso di file, tratta questo come un modello per indicare nomi di file, che viene valutato ottenendo l'elenco dei nomi corrispondenti. In pratica, la valutazione di '**<*.pl>**' restituisce l'elenco dei nomi dei file che terminano con l'estensione '.pl' nella directory corrente. Generalmente è preferibile eseguire un tipo di valutazione del genere in un contesto lista, come nell'esempio seguente:

```
#!/usr/bin/perl

@mioelenco = <*.pl>;
print "@mioelenco\n";
```

In alternativa si può utilizzare la funzione interna ***glob()***, come nell'esempio seguente:


```
#!/usr/bin/perl

@mioelenco = glob ("*.pl");
print "@mioelenco\n";
```

24.9 Funzioni definite dall'utente

Le funzioni definite dall'utente, o subroutine se si preferisce il termine, possono essere collocate in qualunque parte del sorgente Perl. Eventualmente possono anche essere caricate da file esterni. I parametri delle funzioni vengono passati nello stesso modo in cui si fa per le funzioni predefinite, interne a Perl: attraverso una lista di elementi scalari. Le funzioni ottengono i parametri dall'array predefinito '**@**_'. Il valore restituito dalle funzioni è quello dell'ultima istruzione eseguita all'interno della funzione: solitamente si tratta di '**return**' che permette di controllare meglio la cosa.

La sintassi normale per la dichiarazione di una funzione è la seguente. Le parentesi graffe vanno intese in modo letterale e non fanno parte della descrizione del modello sintattico.

```
sub nome { istruzione… }
```

Per la chiamata di una funzione si deve usare la forma seguente:

```
&nome (parametro, …)
```

L'uso della e-commerciale ('**&**') all'inizio del nome è opportuno anche se non è strettamente obbligatorio: permette di evitare ambiguità se il nome della funzione è stato usato per altri tipi di entità all'interno del programma Perl.

Listato 24.63. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/kvE0EHss* , *http://ideone.com/ClHRl* .

```
#!/usr/bin/perl
# I valori da sommare vengono indicati nella riga di
# comando.
$input_1 = ARGV[0];
$input_2 = ARGV[1];

sub somma
{
    return ($_[0] + $_[1]);
}

$totale = &somma ($input_1, $input_2);

print "$input_1 + $input_2 = $totale\n";
```

L'esempio mostrato sopra dovrebbe chiarire il ruolo dell'array '**@**_' all'interno della funzione, come mezzo per il trasporto dei parametri di chiamata.

24.9.1 Chiamata per riferimento e chiamata per valore

L'array '**@**_' è costruito attraverso riferimenti ai parametri utilizzati originariamente nella chiamata. Ciò è sufficiente a fare in modo che modificando il contenuto dei suoi elementi, queste modifiche si riflettano sui parametri di chiamata. Si ha in tal modo quello che si definisce ***chiamata per riferimento***, in cui la funzione è in grado di modificare le variabili utilizzate come parametri.

Naturalmente ciò ha senso solo se i parametri utilizzati sono espressi in forma di variabile e come tali possono essere modificati. Tentare di modificare una costante produce un errore irreversibile.

Dal momento che l'array '**@**_' contiene riferimenti ai dati originali, assegnando all'array un'altra lista di valori non si alterano i dati originali, ma si perde il contatto con quelli. Quindi, non si può assegnare a tale array una lista come modo rapido di variare tutti i parametri della chiamata.

Per gestire elegantemente una funzione che utilizzi il sistema della chiamata per valore, si può fare come nell'esempio seguente:

```
sub miasub
{
    local ($primo, $secondo, $terzo) = @_;
    …
    return …;
}
```

In tal modo, agendo successivamente solo sulle variabili scalari ottenute non si modifica l'array '`@_`' e lo stesso codice diventa più leggibile.

24.9.2 Campo di azione delle variabili

Perl gestisce tre tipi di campi di azione per le variabili (di solito si usa il termine *scope* per fare riferimento a questo concetto). Si tratta di variabili ***pubbliche***, ***private*** e ***locali***.

Le variabili pubbliche sono accessibili in ogni punto del programma, senza alcuna limitazione, a meno che vengano oscurate localmente. Si ottiene una variabile pubblica quando questa viene creata senza specificare nulla di particolare.

```
# Inizializzazione di una variabile pubblica.
$pubblica = "ciao";
```

Una variabile privata è visibile solo all'interno del blocco di istruzioni in cui viene creata e dichiarata come tale. Le funzioni chiamate eventualmente all'interno del blocco, non possono accedere alle variabili private dichiarate nel blocco chiamante. Si dichiara una variabile privata attraverso l'istruzione '**`my`**'.

```
my variabile
```

```
my variabile = valore
```

```
my (variabile1, variabile2, …)
```

Una variabile locale è visibile solo all'interno del blocco di istruzioni in cui viene creata e dichiarata come tale. Le funzioni chiamate eventualmente all'interno del blocco, possono accedere alle variabili locali dichiarate nel blocco chiamante. Si dichiara una variabile locale attraverso l'istruzione '**`local`**'.

```
local variabile
```

```
local variabile = valore
```

```
local (variabile1, variabile2, …)
```

Sia le variabili private, sia quelle locali, permettono di utilizzare un nome già esistente a livello globale, sovrapponendosi temporaneamente a esso. Quelle locali, in particolare, hanno valore anche per le funzioni chiamate all'interno dei blocchi in cui queste variabili sono state dichiarate.

Si dice anche che le variabili private abbiano un campo di azione definito in modo lessicale, mentre quelle locali in modo dinamico: terminata la zona di influenza, le variabili locali vengono rilasciate, mentre quelle private no.

Seguono due esempi di calcolo del fattoriale in modo ricorsivo. In un caso si utilizza una variabile privata, nell'altro una locale. Funzionano entrambi correttamente.

Listato 24.66. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/SEqAr144* , *http://ideone.com/VnwHu* .

```
#!/usr/bin/perl
$input_1 = $ARGV[0];
sub fattoriale
{
    my $valore = $_[0];
    if ($valore > 1)
      {
        return ($valore * &fattoriale ($valore -1));
      }
    else
      {
        return 1;
      }
}

$miofatt = &fattoriale ($input_1);

print "$input_1! = $miofatt\n";
```

Listato 24.67. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/UM6GPQHM* , *http://ideone.com/pnjiq* .

```
#!/usr/bin/perl
$input_1 = $ARGV[0];
sub fattoriale
{
    local $valore = $_[0];
    if ($valore > 1)
      {
        return ($valore * &fattoriale ($valore -1));
      }
    else
      {
        return 1;
      }
}

$miofatt = &fattoriale ($input_1);

print "$input_1! = $miofatt\n";
```

24.10 Variabili contenenti riferimenti

Si è accennato al fatto che una variabile scalare può contenere anche ***riferimenti***, oltre a valori stringa o numerici. Il riferimento è un modo alternativo per puntare a un'entità determinata del programma. La gestione di questi riferimenti da parte di Perl è piuttosto complessa. Qui vengono analizzate solo alcune caratteristiche e possibilità.

Perl gestisce due tipi di riferimenti: diretti (*hard*) e simbolici. Volendo fare un'analogia con quello che accade con i collegamenti dei file system Unix, i primi sono paragonabili ai collegamenti fisici (gli *hard link*), mentre i secondi sono simili ai collegamenti simbolici.

24.10.1 Riferimenti diretti

I riferimenti diretti vengono creati utilizzando l'operatore barra obliqua inversa ('`\`'), come negli esempi seguenti:

```
$rifscalare     = \$mioscalare;
$rifarray       = \@mioarray;
$rifhash        = \%miohash;
$rifcodice      = \&miafunzione;
$rifflusso      = \*MIO_FILE;
```

Esiste anche una forma sintattica alternativa di esprimere i riferimenti: si tratta di indicare il nome dell'entità per la quale si vuole creare il riferimento, preceduto da un asterisco e seguito dalla definizione del tipo a cui questa entità appartiene, tra parentesi graffe.

```
$rifscalare     = *mioscalare{SCALAR};
$rifarray       = *mioarray{ARRAY};
$rifhash        = *miohash{HASH};
$rifcodice      = *miafunzione{CODE};
$rifflusso      = *MIO_FILE{IO};
```

Perl riconosce anche il tipo '**FILEHANDLE**' equivalente a '**IO**', per motivi di compatibilità con il passato.

24.10.2 Riferimenti simbolici

I riferimenti simbolici sono basati sul nome dell'entità a cui si riferiscono, per cui, una variabile scalare contenente il nome dell'oggetto può essere gestita come un riferimento simbolico. Seguono alcuni degli esempi visti nel caso dei riferimenti diretti, in quanto con questo tipo di riferimenti non si possono gestire tutte le situazioni.

```
$rifscalare = 'mioscalare';
$rifarray   = 'mioarray';
$rifhash    = 'miohash';
$rifcodice  = 'miafunzione';
```

Generalmente, l'utilizzo di riferimenti simbolici è sconsigliabile, a meno che ci sia una buona ragione.

24.10.3 Dereferenziazione

Restando in questi termini, a parte il caso dei flussi di file, il modo per ***dereferenziare*** le variabili che contengono i riferimenti è uguale per entrambi i tipi. La forma normale richiede l'utilizzo delle parentesi graffe per delimitare lo scalare. In precedenza si è visto che una variabile scalare può essere indicata attraverso la forma '**${*nome*}**'. Estendendo questo concetto, racchiudendo tra parentesi graffe un riferimento, si ottiene l'oggetto stesso. Per cui:

- `${$rifscalare}`
  equivale a utilizzare '**$mioscalare**';
- `${$rifscalare}[0]`
  equivale a utilizzare '**$mioarray[0]**';
- `${$rifhash}{primo}`
  equivale a utilizzare '**$miohash{primo}**';
- `&{$rifcodice} (1, 7)`
  equivale a utilizzare '**&miafunzione (1, 7)**'.

Sono anche ammissibili altre forme, più espressive o più semplici. La tabella 24.71 riporta alcuni esempi con le forme possibili per dereferenziare gli scalari contenenti dei riferimenti.

Tabella 24.71. Esempi attraverso cui dereferenziare le variabili scalari contenenti dei riferimenti.

${$rifscalare}	**$$rifscalare**	
${$rifscalare}[0]	**$$rifscalare[0]**	**$rifscalare->[0]**
${$rifhash}{primo}	**$$rifhash{primo}**	**$rifhash->{primo}**
&{$rifcodice} (1, 7)	**&$rifcodice (1, 7)**	**$rifcodice-> (1, 7)**

Il caso dei flussi di file è più semplice, in quanto è sufficiente valutare il riferimento, invece del flusso di file vero e proprio. L'esempio seguente dovrebbe chiarire il meccanismo:

```
$rifstdio  = \*STDIO;
$riga = <$rifstdio>;
```

24.10.4 Array multidimensionali

Gli array di Perl hanno una sola dimensione. Per ovviare a questo inconveniente si possono utilizzare elementi che fanno riferimento ad altri array. In pratica, si potrebbe fare qualcosa di simile all'esempio seguente:

```
@primo   = (1, 2);
@secondo = (3, 4);

@mioarray = (\@primo, \@secondo);
```


Qui, l'array '**mioarray**' è in pratica una matrice a due dimensioni rappresentabile nel modo seguente:

$$mioarray = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Per accedere a un elemento singolo di questo array, per esempio al primo elemento della seconda riga (il numero tre), si può usare intuitivamente una di queste due forme:

```
${$mioarray[1]}[0]

$mioarray[1]->[0]
```

In alternativa è concessa anche la forma seguente, più semplice e simile a quella di altri linguaggi:

```
$mioarray[1][0]
```

Una particolarità di Perl sta nella possibilità di definire delle entità anonime. Solitamente si tratta di variabili che non hanno un nome e a cui si accede attraverso uno scalare contenente un riferimento diretto al loro contenuto. Il caso più interessante è dato dagli array, perché questa possibilità permette di definire istantaneamente un array multidimensionale. L'array dell'esempio precedente potrebbe essere dichiarato nel modo seguente:

```
@mioarray = ([1, 2], [3, 4]);
```

La gestione pratica di un array multidimensionale secondo Perl, potrebbe sembrare un po' complessa a prima vista. Tuttavia, basta ricordare che si tratta di array dinamici, per cui, basta assegnare un elemento per dichiararlo implicitamente:

```
@mio_array = ();
…
$mio_array[0] = "ciao";
$mio_array[1] = "come";
$mio_array[2] = "stai";
…
```

Come si vede, viene dichiarato l'array senza elementi, al quale questi vengono inseriti solo successivamente. Così facendo, la dimensione dell'array varia in base all'uso che se ne fa. Con questo criterio si possono gestire anche gli array multidimensionali:

```
@mio_array = ();
…
$mio_array[0] = ();
…
$mio_array[0][0] = "ciao";
$mio_array[0][1] = "come";
$mio_array[0][2] = "stai";
…
```

In questo caso, dopo aver dichiarato l'array '**@mio_array**', senza elementi, viene dichiarato il primo elemento come contenente un altro array vuoto; infine, vengono dichiarati i primi tre elementi di questo sotto-array. Il funzionamento dovrebbe essere intuitivo, anche se si tratta effettivamente di un meccanismo molto complesso e potente.

Di fronte a array multidimensionali di questo tipo, potenzialmente irregolari, si può porre il problema di conoscere la lunghezza di un sotto-array. Volendo usare la tecnica del prefisso '**$#**', si potrebbe fare come nell'esempio seguente, per determinare la lunghezza dell'array contenuto in '**$mio_array[0]**'.

```
$ultimo = $#{$mio_array[0]};
```

24.10.5 Alias

Attraverso l'uso dei riferimenti, è possibile creare un alias di una variabile. Per comprendere questo è necessario introdurre l'uso dell'asterisco. Si osservi questo esempio: se '**$variabile**' rappresenta una variabile scalare, '***variabile**' rappresenta il puntatore alla variabile omonima. In un certo senso, '***variabile**' è equivalente a '**\$variabile**', ma non è proprio la stessa cosa. Si osservino gli assegnamenti seguenti, supponendo che esista già la variabile '**$tua**' e si tratti di uno scalare.

```
*mia = \$tua;
*mia = *tua;
```

I due assegnamenti sono identici, perché in entrambi i casi si assegna a '***mia**' il riferimento alla variabile scalare '**$tua**'. Il risultato di questo è che si può usare la variabile scalare '**$mia**' come alias di '**$tua**'. L'esempio seguente dovrebbe chiarire meglio la cosa.

```
#!/usr/bin/perl
$tua = "ciao";
*mia = \$tua;
print "$mia\n";
```

Quello che si ottiene è l'emissione della stringa '**ciao**', cioè il contenuto della variabile '**$tua**', ottenuto attraverso l'alias '**$mia**'.

Attraverso gli alias è possibile gestire agevolmente il passaggio di parametri per riferimento nelle chiamate delle funzioni. Si osservi l'esempio seguente, in cui una funzione altera il contenuto di un array, senza che questo debba essere dichiarato come variabile globale.

Listato 24.83. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/q82n8gNK* , *http://ideone.com/R3cXT* .

```
#!/usr/bin/perl
sub alterazione_array
{
  local (*a) = $_[0];
  $a[0] = 1;
  $a[1] = 2;
}

local ($b) = ();

$b[0] = 9;
$b[1] = 8;
$b[2] = 7;

&alterazione_array (\@b);

print STDOUT $b[0] . " " . $b[1] . " " . $b[2] . "\n";
```

Eseguendo questo programmino molto semplice, si ottiene la stringa seguente:

```
1 2 7
```

Questo serve a dimostrare che i primi due elementi dell'array sono stati modificati dalla funzione.

24.11 Avvio di Perl

Normalmente è sufficiente rendere eseguibile uno script Perl per fare in modo che il programma '`/usr/bin/perl`' venga eseguito automaticamente per la sua interpretazione. Il programma '`/usr/bin/perl`' permette di utilizzare alcune opzioni, principalmente utili per individuare errori sintattici e problemi di altro tipo.

Opzione	Descrizione
`-c`	Analizza sintatticamente lo script e termina senza eseguirlo.
`-w`	Viene usato assieme a '**-c**' e permette di avere informazioni più dettagliate su problemi eventuali che non sono necessariamente considerabili come errori sintattici.
`-d`	Esegue lo script all'interno di un sistema diagnostico di *debug*.

Segue la descrizione di alcuni esempi.

- `$` **`perl mio.pl`** [*Invio*]

  Avvia il programma Perl '**mio.pl**'. Generalmente si avvia direttamente lo script, ma se questo non è stato reso eseguibile attraverso i permessi, si può ovviare in questo modo.

- `$` **`perl -c mio.pl`** [*Invio*]

  Analizza lo script '**mio.pl**' senza eseguirlo. Se tutto va bene si ottiene l'output seguente:

  ```
  mio.pl syntax OK
  ```


- `$` **`perl -c -w mio.pl`** [*Invio*]

  Come nell'esempio precedente, con l'aggiunta dell'opzione '**-w**', con la quale si ottengono maggiori indicazioni e suggerimenti per migliorare il programma.

- `$` **`perl -d mio.pl`** [*Invio*]

  Avvia il sistema diagnostico per il programma '**mio.pl**'.

24.12 Operatori di delimitazione di stringhe

Nella sezione dedicata agli operatori e alle espressioni rimangono in sospeso gli operatori di delimitazione di stringhe. Nei linguaggi di programmazione tradizionale esiste normalmente il problema di delimitare le stringhe, ovvero le costanti alfanumeriche. Sono già stati mostrati due tipi di delimitatori, gli apici doppi e singoli che hanno un comportamento simile a quello delle shell comuni. In realtà Perl ha una gestione molto più raffinata e generalizzata delle stringhe. Quando il tipo di delimitazione, ovvero il tipo di stringa, lo consente, sono validi alcuni codici di escape. La tabella 24.87 mostra l'elenco di queste sequenze di escape utilizzabili nelle stringhe.

Tabella 24.87. Elenco delle sequenze di escape utilizzabili nelle stringhe delimitate con gli apici doppi.

Escape	Corrispondenza
`\\`	'****'
`\"`	'**"**'
`\$`	'**$**'
`\@`	'**@**'
`\'`	'**'**
`\t`	*<HT>*
`\n`	*<LF>*
`\r`	*<CR>*
`\f`	*<FF>*
`\a`	*<BEL>*
`\e`	*<ESC>*
`\0`***nnn***	Carattere a 8 bit, corrispondente al numero ottale rappresentato da ***nnn***.
`\x`***hh***	Carattere a 8 bit, corrispondente al numero esadecimale rappresentato da due cifre: ***hh***.
`\x{`***h***`}`	Carattere esteso (UTF-8) corrispondente al punto di codifica indicato in esadecimale, composto da un numero qualunque di cifre.
`\[`	Carattere di controllo.
`\l`	Il carattere successivo in minuscolo.
`\u`	Il carattere successivo in maiuscolo.
`\L`	Minuscolo fino al codice '**\E**'.
`\U`	Maiuscolo fino al codice '**\E**'.
`\E`	Conclusione di un modificatore.
`\Q`	Evita l'interpretazione come espressione regolare fino al codice '**\E**'.

La delimitazione dei vari tipi di stringa avviene in una forma tradizionale, attraverso delimitatori che esprimono di per sé il tipo di stringa, oppure attraverso una forma che consente di cambiare tipo di delimitatore:

```
x delim_sinistro stringa delim_destro eventuali_opzioni
```

La sigla che appare inizialmente, ***x*** in questo caso, definisce il tipo di stringa; il delimitatore sinistro e quello destro possono essere parentesi aperte e chiuse di qualunque tipo: tonde, quadre, graffe e angolari, ma si possono utilizzare anche altri simboli, solo che in tal caso, il delimitatore sinistro e quello destro sono uguali.

La tabella 24.109, alla fine di questo gruppo di sezioni, riassume i vari tipi di operatori di delimitazione delle stringhe.

24.12.1 Stringa letterale non interpolata: «q//» o «' '»

La stringa letterale non interpolata è stringa racchiusa normalmente tra apici singoli (è già stata descritta in precedenza). In particolare, restituisce la stringa racchiusa senza effettuare l'interpolazione delle eventuali variabili e dei simboli di escape che dovesse incorporare, a eccezione di '`\'`' e '`\\`'. Si può esprimere in due modi:

```
'stringa'
```

```
qdelim_sinistro stringa delim_destro
```

Seguono alcuni esempi:

```
$miavar = 'Stringa tradizionale che non interpola';
```

```
$miavar = q|Una stringa che "contiene 'apici' di ogni tipo".|;
```

```
$miavar = q(Sembra una funzione, ma non lo è);
```

```
$miavar = q{Le variabili non vengono interpolate. $ciao};
```

24.12.2 Stringa letterale interpolata: «qq//» o «" "»

La stringa letterale interpolata è racchiusa normalmente tra apici doppi (è già stata descritta in precedenza). In particolare, restituisce la stringa racchiusa interpolando le variabili e i simboli di escape che dovesse incorporare. Si può esprimere in due modi:

```
"stringa"
```

```
qqdelim_sinistro stringa delim_destro
```

Seguono alcuni esempi:

```
$miavar = "Stringa tradizionale che interpola";
```

```
$miavar = qq|Una stringa che \"contiene 'apici' di ogni tipo\".|;
```

```
$miavar = qq(Sembra una funzione, ma non lo è);
```

```
$ciao = "Saluti!";
$miavar = qq{Le variabili vengono interpolate. $ciao};
```

24.12.3 Comando di sistema: «qx//» o «` `»

La stringa che rappresenta un comando di sistema deve essere valutata e successivamente eseguita in qualità di comando dal sistema operativo. Questo tipo di stringa è racchiuso normalmente tra apici singoli inversi, come avviene nelle shell comuni. Il contenuto della stringa viene interpolato prima dell'esecuzione del comando. La valutazione della stringa si traduce nell'output emesso attraverso lo standard output dal comando stesso. Si può esprimere in due modi:

```
`stringa`
```

```
qxdelim_sinistro stringa delim_destro
```

Seguono alcuni esempi:

```
$miadata = `date`;
```

```
$mioelenco = qx(ls);
```

```
$opzioni = '-l'
$mioelenco = qx{ls $opzioni};
```


24.12.4 Lista di parole: «qw//»

La stringa racchiusa in questo tipo di delimitazione, non viene interpolata, ma semplicemente restituita in forma di ***lista di parole***. In pratica, tutto ciò che risulta separato da spazi (spazi veri e propri, caratteri di tabulazione e codici di interruzione di riga) viene estratto e inserito in una lista di elementi. Si può esprimere solo nel modo seguente:

```
qwdelim_sinistro stringa delim_destro
```

Seguono alcuni esempi validi:

```
@mialista = qw/ciao come stai/;
```

```
@mialista = qw(uno      due     tre);
```

```
@mialista = qw(alfa     bravo   charlie
delta   echo    foxtrot golf    hotel
india   kilo    lima);
```

24.12.5 Modello di confronto: «m//» o «//»

Il modello di confronto non restituisce alcunché e serve per essere paragonato a un'altra stringa. Può essere usato in un contesto scalare o lista. Nel primo caso serve a determinare se esiste una corrispondenza con il modello o meno. Nel secondo caso, viene sempre paragonato a un'altra stringa, ma il risultato di questo abbinamento è una lista di elementi.

Il modello si esprime in forma di espressione regolare, con delle particolarità che derivano dal tipo di delimitatori utilizzati e dal fatto che prima di valutare l'espressione regolare viene eseguita un'interpolazione. Si può esprimere in due modi.

```
/stringa/opzioni
```

```
mdelim_sinistro stringa delim_destro modificatori
```

I modificatori si esprimono con una serie di lettere, o nulla se non è necessario. La tabella 24.102 ne riporta l'elenco.

Tabella 24.102. Elenco dei modificatori utilizzabili con l'operatore di delimitazione '`m`'.

Modificatore	Descrizione
`i`	Il confronto avviene ignorando la differenza tra maiuscole e minuscole.
`m`	Le stringhe vengono trattate come righe multiple (riguarda '`^`' e '`$`').
`s`	Tratta le stringhe come una riga singola (riguarda '`.`').
`x`	Permette l'inserzione di spazi e commenti che non vengono interpretati.
`g`	Confronta in modo globale, cioè trova tutte le occorrenze.
`o`	Interpreta il modello (e di conseguenza lo interpola) solo la prima volta.

L'utilizzo delle espressioni regolari nelle istruzioni Perl è ciò che generalmente rende il sorgente di un programma piuttosto confuso. Se si devono utilizzare intensivamente le espressioni regolari sarebbe opportuno approfondirne il funzionamento e l'utilizzo di questo tipo di delimitatori, per trovare un modo meno complicato del solito di scrivere queste espressioni. Il primo punto su cui si può intervenire è la scelta dei simboli di delimitazione. La forma tradizionale prevede l'uso della barra obliqua normale, cosa però che crea problemi quando si vuole utilizzare questo simbolo all'interno dell'espressione stessa. Infatti, i simboli usati come delimitazione non possono essere utilizzati nell'espressione regolare senza la tecnica della protezione per mezzo del prefisso '`\`'.

Segue la descrizione di alcuni esempi.

```
#!/usr/bin/perl

$miafrase = 'Ciao, come stai?';
if ($miafrase =~ /ciao/i)
  {
    print "Ciao!\n";
  }
```

In questo esempio, il modello '`/ciao/i`' combacia con una parte della frase, facendo sì che la condizione si avveri.

```
#!/usr/bin/perl

$mioelenco = `ls`;
if ($mioelenco =~ /.*\.pl/)
  {
    print "Ci sono programmi Perl in questa directory.\n";
  }
```

In questo esempio, viene letto il contenuto della directory corrente e posto nella variabile '**`$mioelenco`**'. Successivamente viene verificato se in quell'elenco si trova qualcosa che termina con '`.pl`'. Dal momento che il punto ha un significato nelle espressioni regolari, per poterlo includere si è posta anteriormente una barra obliqua inversa.

24.12.6 Modello di sostituzione: «s//»

Definisce un modello di confronto con una stringa, assieme a una stringa di sostituzione per la parte che corrisponde al modello. Se il confronto non viene fatto attraverso gli operatori '**`=~`**' oppure '**`!~`**', si intende che l'abbinamento avvenga con il contenuto della variabile '**`$_`**'. Ha luogo l'interpolazione.

L'abbinamento per la sostituzione può avvenire solo in un contesto scalare. Il modello si esprime in forma di espressione regolare. La sintassi può essere espressa in due modi, a seconda del tipo di delimitatori utilizzati.

`s`***delim_sx stringa delim_dx delim_sx rimpiazzo delim_dx modificatori***

`s`***delim stringa delim rimpiazzo delim modificatori***

Il primo tipo di sintassi si adatta al caso in cui si usino parentesi per delimitare le stringhe del modello e del rimpiazzo, il secondo tipo si riferisce all'uso di altri simboli che non sono utilizzati in coppia.

I modificatori si esprimono con una serie di lettere, o nulla se ciò non è necessario. La tabella 24.105 ne riporta l'elenco.

Tabella 24.105. Elenco dei modificatori utilizzabili con l'operatore di delimitazione '**`s`**'.

Modificatore	Descrizione
`i`	Il confronto avviene ignorando la differenza tra maiuscole e minuscole.
`m`	Le stringhe vengono trattate come righe multiple (riguarda '`^`' e '`$`').
`s`	Tratta le stringhe come una singola riga (riguarda '`.`').
`x`	Permette l'inserzione di spazi e commenti che non vengono interpretati.
`g`	Confronta in modo globale, cioè trova tutte le occorrenze.
`o`	Interpreta il modello (e di conseguenza lo interpola) solo la prima volta.
`e`	Valuta la parte destra come un'espressione.

Segue la descrizione di alcuni esempi.

- `$path =~ s|/usr/bin|/usr/local/bin|`

  Sostituisce la prima occorrenza di '`/usr/bin`' nella variabile '**`$path`**' con '`/usr/local/bin`'. Per delimitare il modello e la stringa di sostituzione sono state usate le barre verticali, per evitare ambiguità con le barre oblique delle directory.

- `$path =~ s{/usr/bin}{/usr/local/bin}`

  Esattamente come nell'esempio precedente, ma questa volta sono state usate le parentesi graffe.

24.12.7 Sostituzione di caratteri: «tr//» o «y//»

La stringa racchiusa in questo tipo di delimitatore, definisce un modello di sostituzione di una serie di caratteri in un'altra. Si applica al contenuto di una variabile scalare utilizzando l'operatore '**`=~`**' oppure '**`!~`**', altrimenti si intende la variabile '**`$_`**'. Restituisce il numero di trasformazioni eseguite. Non ha luogo l'interpolazione.

L'abbinamento per la sostituzione può avvenire solo in un contesto scalare. Il modello si esprime in forma di espressione regolare. La sintassi può essere espressa nei modi seguenti, a seconda che si voglia utilizzare l'identificatore '**`tr`**' o '**`y`**' e a seconda del tipo di delimitatori utilizzati.

`tr`***delim_sx car_da_sost delim_dx delim_sx rimpiazzo delim_dx modificatori***

`tr`***delim car_da_sostituire delim rimpiazzo delim modificatori***

`y`***delim_sx car_da_sost delim_dx delim_sx rimpiazzo delim_dx modificatori***

`y`***delim car_da_sostituire delim rimpiazzo delim modificatori***

I modificatori si esprimono con una serie di lettere, o nulla se ciò non è necessario. La tabella 24.108 ne riporta l'elenco.

Tabella 24.108. Elenco dei modificatori utilizzabili con l'operatore di delimitazione '**`tr`**'.

Modificatore	Descrizione
`c`	Cerca gli elementi che **non** sono elencati nel gruppo da sostituire.
`d`	Cancella i caratteri trovati e non rimpiazzati.
`s`	Fonde insieme i caratteri doppi che sono stati ritrovati.

Tabella 24.109. Elenco riassuntivo dei tipi di operatori di stringa. Le parentesi graffe rappresentano la posizione dei delimitatori.

Formato normale	Formato generico	Significato	Interpolazione
`' '`	`q{}`	Stringa letterale.	NO
`" "`	`qq{}`	Stringa letterale.	SÌ
`` ` ` ``	`qx{}`	Comando di sistema.	SÌ
	`qw{}`	Lista di parole.	NO
`/ /`	`m{}`	Modello di confronto.	SÌ
	`s{}{}`	Modello di sostituzione.	SÌ
	`tr{}{}`	Traslazione di caratteri.	SÌ

Segue la descrizione di alcuni esempi:

- `$miavar =~ tr/A-Z/a-z/;`

  Converte in minuscolo il contenuto della variabile (a parte le vocali accentate).

- `$contatore = ($miavar =~ tr/0-9//);`

  Conta i caratteri numerici contenuti nella variabile '**`$miavar`**'.

24.13 Espressioni regolari

Le espressioni regolari possono essere considerate l'elemento più potente e più difficile di Perl. Purtroppo non esiste una definizione e uno standard universale delle espressioni regolari, così, per ogni applicazione che ne fa uso occorre studiarne le particolarità.

In questa sezione si descrive solo parte delle potenzialità di Perl con le espressioni regolari. Per conoscerne i dettagli è necessario consultare la pagina di manuale *perlre(1)*. Può essere conveniente anche la lettura delle sezioni 23.1 e 23.3.

24.13.1 Modificatori

Perl utilizza le espressioni regolari con gli operatori di stringa '**m{}**' e '**s{}{}**'. Con questi è possibile utilizzare delle opzioni finali, ovvero dei modificatori, che alterano le regole delle espressioni regolari. La tabella 24.112 mostra l'elenco dei modificatori più comuni.

Tabella 24.112. Elenco dei modificatori utilizzabili in generale in coda alle espressioni regolari di Perl.

Modificatore	Descrizione
`i`	Il confronto avviene ignorando la differenza tra maiuscole e minuscole.
`m`	Le stringhe vengono trattate come righe multiple (riguarda '**^**' e '**$**').
`s`	Tratta le stringhe come una riga singola (riguarda '**.**').
`x`	Permette l'inserzione di spazi e commenti che non vengono interpretati.

24.13.2 Metacaratteri

In generale, i caratteri utilizzati in un'espressione regolare, che non abbiano un significato speciale, corrispondono a loro stessi nella stringa di comparazione. Ciò è come dire che la comparazione seguente è valida:

```
'Ciao' =~ /Ciao/
```

I ***metacaratteri*** di un'espressione regolare sono dei simboli che hanno un significato diverso rispetto ai caratteri utilizzati per rappresentarli. La tabella 24.114 mostra l'elenco dei metacaratteri più comuni.

Tabella 24.114. Elenco dei metacaratteri standard utilizzati in Perl.

Metacarattere	Descrizione
`\`	Protegge il carattere successivo da un'interpretazione diversa da quella letterale.
`^`	Corrisponde all'inizio di una riga.
`.`	Corrisponde a un carattere qualunque.
`$`	Corrisponde alla fine di una riga.
`|`	Indica due possibilità alternative alla sua sinistra e alla sua destra.
`( )`	Definiscono un raggruppamento.
`[ ]`	Definiscono una classe di caratteri.

La barra obliqua inversa protegge il carattere successivo da un'interpretazione diversa da quella letterale, quando la sequenza '***x***' (***x*** rappresenta qui un carattere qualunque) non rappresenta già un metacarattere. In pratica, se '***x***' non ha un significato particolare, rappresenta semplicemente '***x***' in modo letterale.

L'accento circonflesso ('**^**') corrisponde generalmente all'inizio di una riga; nello stesso modo, il simbolo dollaro ('**$**') rappresenta la fine di una riga. Questi metacaratteri rappresentano in pratica la stringa nulla di inizio e di fine di una riga. Se la stringa da analizzare è composta da più righe terminate dal codice di interruzione di riga, è possibile fare in modo che '**^**' e '**$**' corrispondano all'inizio e alla fine di queste righe virtuali utilizzando il modificatore '**m**'.

Il punto rappresenta un carattere singolo, con l'esclusione del codice di interruzione di riga a meno che sia stato utilizzato il modificatore '**s**'.

Perl aggiunge a quelli standard una serie di metacaratteri rappresentati dalla tabella 24.115; nella stessa tabella vengono anche elencate le classi di caratteri, argomento che viene descritto nella sezione successiva.

Tabella 24.115. Elenco dei metacaratteri speciali di Perl e delle classi POSIX disponibili.

Metacarattere	Compatibilità POSIX	Corrispondenza
`\w`	`[[:word:]]`	Un carattere alfanumerico (lettere e numeri) compreso il trattino basso.
`\W`	`[^[:word:]]`	Un carattere non alfanumerico (l'opposto di '**\w**').
`\s`	`[[:space:]]`	Uno spazio lineare (spazio o tabulazione).
`\S`	`[^[:space:]]`	Qualunque carattere che non sia uno spazio lineare.
`\d`	`[[:digit:]]`	Un carattere numerico.
`\D`	`[^[:digit:]]`	Un carattere non numerico.
`\b`		La stringa nulla prima o dopo una sequenza di caratteri corrispondenti a '**\w**'.
`\B`		La stringa nulla interna a una sequenza di caratteri corrispondenti a '**\w**'.
`\A`		L'inizio di una stringa.
`\Z`		La fine di una stringa (eventualmente prima di un *new-line* finale).
	`[[:alpha:]]`	Un carattere alfabetico.
	`[[:alnum:]]`	Un carattere alfanumerico (non corrisponde esattamente a '**\w**').
	`[[:ascii:]]`	Un carattere ASCII.
	`[[:blank:]]`	Uno spazio lineare che comprende altri caratteri oltre a '**\s**'.
	`[[:cntrl:]]`	Un carattere di controllo, inteso come carattere che non ha una rappresentazione grafica.
	`[[:graph:]]`	Un carattere qualunque che abbia una rappresentazione grafica: alfabetico, numerico, di punteggiatura e qualunque altro simbolo.
	`[[:print:]]`	Un carattere qualunque che abbia una rappresentazione grafica: alfabetico, numerico, di punteggiatura e qualunque altro simbolo, compreso lo spazio.
	`[[:punct:]]`	Un carattere di punteggiatura.
	`[[:xdigit:]]`	Equivale a '**[0-9A-Fa-f]**', ovvero un carattere che si può usare per rappresentare un valore esadecimale.
	`[[:lower:]]`	Un carattere alfabetico minuscolo.
	`[[:upper:]]`	Un carattere alfabetico maiuscolo.

Inoltre, per complicare ulteriormente le cose, le espressioni regolari di Perl vengono trattate come se fossero racchiuse tra apici doppi, cioè vengono interpolate prima di essere valutate come espressioni regolari. Questo significa che le variabili vengono espanse e vengono riconosciuti anche altri simboli che in pratica potrebbero essere considerati come dei metacaratteri aggiuntivi. Si tratta di '**\n**', '**\t**' e altri come già indicato nella tabella 24.87 all'inizio del capitolo.

24.13.3 Classi di caratteri

Un modello racchiuso tra parentesi quadre rappresenta un solo carattere in base a quanto indicato nelle parentesi.

Una fila di caratteri racchiusa tra parentesi quadre corrisponde a un carattere qualunque tra quelli indicati; se all'inizio di questa fila c'è

l'accento circonflesso, si ottiene una corrispondenza con un carattere qualunque diverso da quelli della fila. Per esempio, l'espressione regolare '**[0123456789]**' corrisponde a una cifra numerica qualunque, mentre '**[^0123456789]**' corrisponde a un carattere qualunque purché non sia una cifra numerica.

All'interno delle parentesi quadre, invece che indicare un insieme di caratteri, è possibile indicarne un intervallo mettendo il carattere iniziale e finale separati da un trattino ('**-**'). I caratteri che vengono rappresentati in questo modo dipendono dalla codifica che ne determina la sequenza. Per esempio, l'espressione regolare '**[9-A]**' rappresenta un carattere qualsiasi tra: '**9**', '**:**', '**;**', '**<**', '**=**', '**>**', '**?**', '**@**' e '**A**', perché così è la sequenza ASCII.

Questa definizione corrisponde in parte a quella di '**grep**' GNU, in particolare si deve tenere presente che all'interno delle parentesi quadre, '**\b**' corrisponde al carattere *<BS>*.

24.13.4 Qualificatori: operatori di ripetizione

Attraverso altri simboli è possibile indicare la ripetizione di un carattere determinato o di un raggruppamento. La tabella 24.116 mostra l'elenco di queste notazioni e il loro significato.

Tabella 24.116. Operatori di ripetizione, o qualificatori, nelle espressioni regolari di Perl.

Codifica	Corrispondenza
x*	Nessuna o più volte ***x***. Equivalente a '***x*{0,}**'.
x?	Nessuna o al massimo una volta ***x***. Equivalente a '***x*{0,1}**'.
x+	Una o più volte ***x***. Equivalente a '***x*{1,}**'.
***x*{*n*}**	Esattamente ***n*** volte ***x***.
***x*{*n*,}**	Almeno ***n*** volte ***x***.
***x*{*n*,*m*}**	Da ***n*** a ***m*** volte ***x***.
x*?	Equivale al minimo di '***x****'.
x??	Equivale al minimo di '***x*?**'.
x+?	Equivale al minimo di '***x*+**'.
***x*{*n*}?**	Equivale al minimo di '***x*{*n*}**', ovvero allo stesso '***x*{*n*}**'.
***x*{*n*,}?**	Equivale al minimo di '***x*{*n*,}**'.
***x*{*n*,*m*}?**	Equivale al minimo di '***x*{*n*,*m*}**'.

Dalla tabella si può osservare la presenza di qualificatori insoliti che terminano con un punto interrogativo. Un modello espresso in forma di espressione regolare può corrispondere a una stringa in diversi modi. Generalmente, la corrispondenza dei qualificatori avviene nel modo più ampio possibile. Se è necessario che la corrispondenza avvenga nel modo più ristretto possibile, occorre utilizzare i qualificatori che terminano con il punto interrogativo. Per esempio, di seguito si vedono alcune corrispondenze valide e le zone delle stringhe originali in cui i modelli combaciano.

```
"CIAO" =~ /\w+/
 ^--^

"Ciao, come stai?" =~ /\s/
      ^

"Ciao, come stai? Io sto bene." =~ /\s.*\s/
      ^-----------------^

"Ciao, come stai? Io sto bene." =~ /\s.*?\s/
      ^----^
```


24.13.5 Raggruppamenti

Una o più parti di un'espressione regolare possono essere raggruppate attraverso l'uso delle parentesi tonde. Ciò permette di abbinare tali raggruppamenti ai qualificatori (gli operatori di ripetizione), oppure permette di estrarre ciò che corrisponde al segmento racchiuso tra parentesi, o di potervi fare riferimento. Per esempio, l'espressione '**\s(come\s)+.*\s**' è valida per tutte le stringhe seguenti.

```
"Ciao, come stai? Io sto bene."
"Ciao, come come stai? Io sto bene."
"Ciao, come come come stai? Io sto bene."
…
```

All'interno della stessa espressione regolare è possibile fare riferimento a una corrispondenza parziale contenuta in un raggruppamento. Per farlo si utilizza il metacarattere '***n***', dove ***n*** è una sola cifra numerica. In pratica, '**\1**' corrisponde al primo raggruppamento, '**\2**' corrisponde al secondo, proseguendo così, di seguito, fino al nono.

Per esempio, '**(0|0x0)\d*\s\1\d***' è valida per '**0x0123 0x0456**', ma non per '**0x0123 0456**'. Infatti, si fa riferimento alla corrispondenza, non al modello che potrebbe essere ripetuto agevolmente.

Perl permette di utilizzare queste corrispondenze anche al di fuori delle espressioni regolari. Per questo però non si può più utilizzare la notazione '***n***', ma occorre invece '**$*n***'. In pratica si tratta di variabili predefinite che vengono generate per l'occasione.

```
s/^(\w+)\s+(\w+)/$2 $1/
```

L'esempio appena mostrato inverte le prime due parole ed elimina gli spazi superflui tra le due. Un altro esempio interessante è il seguente, in cui si estrae la data da una stringa, per gestirla all'interno del programma:

```
if ($miadata =~ m|Data:\s+(\d\d)/(\d\d)/(\d{2,4})|)
  {
    $giorno = $1;
    $mese = $2;
    $anno = $3;
  }
```

Come si può vedere, i delimitatori dell'espressione regolare sono stati sostituiti con le barre verticali, in modo da poter utilizzare le barre oblique per l'espressione stessa senza troppi problemi.

24.14 Gestione generale dei file

Prima di poter accedere in qualunque modo a un file, occorre che questo sia stato aperto all'interno del programma, il quale, da quel punto in poi, vi fa riferimento attraverso il flusso di file.

Per una convenzione diffusa, i nomi attribuiti ai flussi di file sono sempre composti da lettere maiuscole, cosa che facilita il loro riconoscimento all'interno di un sorgente Perl.

Oltre ai file su disco, esistono tre file particolari: standard input, standard output e standard error. Questi risultano sempre già aperti e ai flussi di file corrispondenti si fa riferimento attraverso tre nomi predefiniti: '**STDIN**', '**STDOUT**' e '**STDERR**'.

In condizioni normali, i file si intendono contenere una codifica a 8 bit; mentre è possibile specificare esplicitamente che questi utilizzano la codifica UTF-8.

24.14.1 Apertura

Quando è necessario aprire un file, cioè quando non si tratta dei flussi di file predefiniti, si utilizza la funzione ***open()***.

```
open flusso, file
```

```
open flusso, modalità, file
```

La funzione utilizza quindi due o tre argomenti: il nome del flusso di file, il nome effettivo del file (che può contenere l'indicazione del percorso necessario a raggiungerlo) ed eventualmente la modalità di apertura. Nel primo caso, si intende che il file contiene, o deve contenere, una codifica a 8 bit, mentre nel secondo si può specificare in modo preciso la codifica.

L'esempio seguente apre il file '`mio_file`' che si trova nella directory corrente e gli abbina il flusso di file '**`MIO_FILE`**':

```
open MIO_FILE, 'mio_file';
```

Con l'apertura del file si deve definire anche in che modo si intende accedervi. Fondamentalmente si distingue tra lettura e scrittura, ma in realtà si presentano anche altre sfumature. Per poter informare la funzione del modo in cui si intende aprire il file, la stringa che viene utilizzata per indicare il nome del file su disco può contenere dei simboli aggiuntivi che servono proprio per questo, oppure si usa l'argomento ulteriore che si colloca prima del nome del file. In presenza di soli due argomenti, tali simboli vanno posti quasi sempre di fronte al nome e possono essere spaziati da questo in modo da facilitarne la lettura:

`open` *riferimento*`, "<`*file*`"` `open` *riferimento*`, "<:`*codifica*`", "`*file*`"`	se non si utilizza alcun simbolo, oppure se si pone '**<**', si ottiene l'apertura in lettura (input);
`open` *riferimento*`, ">`*file*`"` `open` *riferimento*`, ">:`*codifica*`", "`*file*`"`	se si utilizza il simbolo '**>**' si intende aprire il file in scrittura (output), troncando inizialmente il file;
`open` *riferimento*`, ">>`*file*`"` `open` *riferimento*`, ">>:`*codifica*`", "`*file*`"`	se si utilizza il simbolo '**>>**' si intende aprire il file in scrittura in aggiunta (*append*).

A questa simbologia si può aggiungere il segno '**+**' in modo da permettere anche l'altro tipo di accesso non dichiarato, per cui:

`open` *riferimento*`, "+<`*file*`"` `open` *riferimento*`, "+<:`*codifica*`", "`*file*`"`	rappresenta un accesso in lettura e scrittura;
`open` *riferimento*`, "+>`*file*`"` `open` *riferimento*`, "+>:`*codifica*`", "`*file*`"`	rappresenta un accesso in scrittura e lettura, ma la prima azione è quella di troncare il file annullando il suo contenuto precedente;
`open` *riferimento*`, "+>>`*file*`"` `open` *riferimento*`, "+>>:`*codifica*`", "`*file*`"`	rappresenta un accesso in aggiunta e lettura.

In generale, un file aperto in lettura e scrittura attraverso il simbolo '**+<**' permette anche l'allungamento del file stesso. Il pezzo di codice seguente mostra l'apertura di un file in aggiunta e l'inserimento al suo interno di una riga contenente una frase di saluto.

```
open MIO_FILE, ">> /home/tizio/mio_file";
…
print MIO_FILE "ciao a tutti\n";
```

L'esempio seguente è una variante in cui si dichiara espressamente l'utilizzo della codifica UTF-8 e si inseriscono alcune lettere greche, specificando i punti di codifica U+03B1, U+03B2, U+03B3:

```
open MIO_FILE, ">>:utf8", "/home/tizio/mio_file";
…
print MIO_FILE "alfa, beta, gamma: ";
print MIO_FILE "\x{03B1}, \x{03B2}, \x{03B3}\n";
```

Eventualmente, si può dichiarare che la codifica deve essere di un certo tipo, attraverso l'istruzione seguente:

```
use open ":codifica"
```


Nello stesso modo in cui si possono gestire i file su disco, si può accedere a un condotto, cioè una sequenza di programmi che ricevono dati dal loro standard input e ne emettono attraverso lo standard output. Per ottenere questo, al posto di indicare un file su disco si mette una riga di comando che si vuole sia eseguita, preceduta o terminata con la consueta barra verticale: se si trova all'inizio, significa che si vuole scrivere inviando dati attraverso lo standard input del condotto; se si trova alla fine, significa che si vuole leggere attingendo dati dallo standard output del condotto.

```
open MIAPIPE, "| sort > /home/tizio/mio_file";
```

L'esempio appena mostrato apre un condotto in scrittura. Ciò che viene ricevuto dal condotto viene ordinato e registrato nel file '`/home/tizio/mio_file`'.

```
open MIAPIPE, "ls -l |";
```

L'esempio precedente apre un condotto in lettura in modo da poter elaborare il risultato del comando '**`ls -l`**'.

In questi casi non si può dichiarare la codifica nell'istruzione '**`open`**', pertanto qui conviene usare l'istruzione '**`use open`**'. Ecco gli stessi esempi appena presentati, in cui si dichiara la scrittura e la lettura secondo la codifica UTF-8:

```
use open ":utf8";
open MIAPIPE, "| sort > /home/tizio/mio_file";
```

```
use open ":utf8";
open MIAPIPE, "ls -l |";
```

Naturalmente, occorre considerare che l'istruzione '**`use open`**' rimane valida per tutti i file che vengono aperti successivamente, fino a quando se ne appare un'altra che ne cambia la modifica.

24.14.2 Codifica di file già aperti

Dal momento che i flussi standard (standard input, standard output e standard error) risultano già aperti in modo predefinito, esiste la possibilità di dichiarare la codifica di file dopo che questi sono già stati aperti:

```
binmode flusso, ":codifica"
```

Per esempio, per dichiarare che tutti i flussi standard usano la codifica UTF-8, bastano le istruzioni seguenti:

```
binmode STDIN, ":utf8";
binmode STDOUT, ":utf8";
binmode STDERR, ":utf8";
```

24.14.3 Chiusura

Un file aperto che non serve più deve essere chiuso. Ciò si ottiene attraverso la funzione ***close()*** indicando semplicemente il flusso di file da chiudere.

```
close flusso
```

L'apertura di un file può essere fatta anche se questo risulta già aperto, per cui non è strettamente necessario chiudere un file prima di riaprirlo.

24.15 Condivisione dei file

In presenza di un sistema operativo in multiprogrammazione, tanto più se anche multiutente, si pone il problema della gestione degli accessi simultanei ai file. In pratica occorre gestire un sistema di blocchi, o di semafori, che impediscano le operazioni di scrittura simultanea da parte di processi indipendenti.

Infatti, la lettura simultanea di un file da parte di più programmi non ha alcun effetto collaterale, mentre la modifica simultanea può tradursi anche in un danneggiamento dei dati. Per questo, quando un file deve essere modificato, è importante che venga impedito ad altri programmi di fare altrettanto, almeno per il tempo necessario a concludere l'operazione.

24.15.1 Blocco dei file

Il modo più semplice per impedire che un file possa essere modificato da un altro processo, è quello di bloccarlo (*lock*), per il tempo necessario a compiere le operazioni che si vogliono fare in modo esclusivo.

Teoricamente, il blocco potrebbe limitarsi solo a una porzione del file, ma questo implica un'organizzazione condivisa anche dagli altri processi, in modo che sia ben definita l'estensione di questo blocco. In pratica, ci si limita quasi sempre a eseguire un blocco totale del file, rilasciando il blocco subito dopo la modifica che si vuole effettuare.

Il blocco e lo sblocco del file si ottiene generalmente con la funzione ***flock()*** su un file già aperto. La funzione richiede l'indicazione del flusso di file e del tipo di operazione che si vuole compiere.

```
flock flusso, operazione
```

Per la precisione, il tipo di operazione si esprime attraverso un numero il cui valore dipende dal sistema operativo utilizzato effettivamente. Per evitare di doversi accertare di quale valore sia corretto per il proprio sistema, è possibile acquisire alcune macro attraverso l'istruzione seguente:

```
use Fcntl ':flock';
```

In questo modo, l'operazione può poi essere indicata attraverso i nomi: '**LOCK_SH**', '**LOCK_EX**', '**LOCK_NB**' e '**LOCK_UN**'.

Il blocco del file può essere richiesto in modo da mettere in pausa il programma fino a quando si riesce a ottenere il blocco, oppure no. Nel secondo caso, il programma deve essere in grado di riconoscere il fallimento dell'operazione e di comportarsi di conseguenza. Il blocco con attesa deve essere utilizzato con prudenza, perché può generare una situazione di stallo generale: il processo A apre e blocca il file X, il processo B apre e blocca il file Y e successivamente tenta anche con il file X che però è occupato; a questo punto anche il processo A tenta di aprire il file Y senza avere rilasciato il file X; infine i due processi si sono bloccati a vicenda.

Il blocco esclusivo di un file si ottiene con il tipo di operazione '**LOCK_EX**'; se si vuole evitare l'attesa dello sblocco da parte di un altro processo si deve aggiungere il valore di '**LOCK_NB**'. Lo sblocco di un file si ottiene con il tipo di operazione '**LOCK_UN**'.

Segue la descrizione di alcuni esempi.

- ```
  use Fcntl ':flock';   # importa le costanti LOCK_…
  …
  open (ELENCO, ">> /home/tizio/mioelenco");
  flock (ELENCO, LOCK_EX);
  …
  flock (ELENCO, LOCK_UN);
  ```

  Vengono eseguite le operazioni seguenti:

  – si caricano le costanti di definizione dei tipi di blocco attraverso l'istruzione '**use Fcntl ':flock';**';
  – si apre il file '`/home/tizio/mioelenco`' in aggiunta;
  – si blocca il file in modo esclusivo;
  – si compiono alcune operazioni che non sono indicate;
  – si rilascia il blocco.

- ```
  use Fcntl ':flock';   # importa le costanti LOCK_…
  …
  open (ELENCO, ">> /home/tizio/mioelenco");
  if (flock (ELENCO, (LOCK_EX)+(LOCK_NB)))
    {
      …
      flock (ELENCO, LOCK_UN);
    }
  else
    {
      print STDOUT "Il file è impegnato.\n";
    }
  ```


Si tratta di una variante dell'esempio precedente, in cui si richiede un blocco esclusivo senza attesa. Se il blocco ha successo, si procede, altrimenti viene segnalata la presenza del blocco da parte di un altro processo.

Per qualche motivo, se si vuole sommare il valore della macro ***LOCK_EX*** assieme a quello di qualche altra, è necessario racchiuderla tra parentesi, come si vede nell'esempio. Probabilmente questo dipende dal modo in cui il valore viene generato. Per uniformità, nell'esempio si mostra racchiusa tra parentesi anche la macro ***LOCK_NB***. Volendo verificare questa anomalia, basta provare ad assegnare a una variabile la somma di queste o di altre macro, visualizzando poi il risultato; se si prova una cosa del tipo '**`$pippo = LOCK_EX+LOCK_NB;`**', senza parentesi, e poi si visualizza il contenuto di '**`$pippo`**', si ottiene solo il valore due, mentre dovrebbe essere un sei!

24.16 I/O con i file

Le operazioni di I/O con i file richiedono la conoscenza del modo in cui si esegue la lettura, la scrittura e lo spostamento, del puntatore interno a un flusso di file. Fortunatamente, Perl gestisce tutto in modo piuttosto trasparente, soprattutto per ciò che riguarda la lettura. È il caso di ricordare che queste operazioni si compiono su file già aperti, di conseguenza si fa riferimento a loro tramite il flusso corrispondente.

24.16.1 Lettura

La lettura di un flusso di file riferito a un file di testo è un'operazione molto semplice, basta utilizzare le parentesi angolari per ottenere la valutazione dello stesso che si traduce nella restituzione di una riga, nel caso di contesto scalare, o di tutto il file, nel caso di un contesto lista. L'esempio seguente restituisce una riga, a partire dalla posizione del puntatore del file fino al codice di interruzione di riga incluso, spostando in avanti il puntatore del file:

```
$riga = <MIOHANDLE>;
```

Per questo, dopo un'operazione di questo tipo, si esegue un ***chop()*** o un ***chomp()***, in modo da eliminare il codice di interruzione di riga finale.

```
chomp $riga;
```

In alternativa, l'istruzione seguente restituisce tutto il file suddiviso in righe terminanti con il codice di interruzione di riga:

```
@file = <MIOHANDLE>;
```

In pratica, l'array viene popolato con tanti elementi quante sono le righe del file. Anche in questo caso si può eseguire un ***chop()*** o un ***chomp()***, che intervenga su ogni elemento dell'array:

```
chomp (@file);
```

La valutazione di un flusso di file in questo modo, quando il puntatore del file ha superato la fine del file, restituisce un valore indefinito che può essere utilizzato per controllare un ciclo di lettura. L'esempio seguente mostra in modo molto semplice come un ciclo '**while**' possa controllare la lettura di un flusso di file terminando quando questo ha raggiunto la conclusione.

```
while ($riga = <MIOHANDLE>)
  {
    …
  }
```

24.16.2 Scrittura

La scrittura di un file avviene generalmente attraverso la funzione ***print()*** che inizia a scrivere a partire dalla posizione attuale del puntatore del file stesso.

```
print flusso lista
```

```
print lista
```

Se non viene specificato un flusso di file, tutto viene emesso attraverso lo standard output, oppure attraverso quanto specificato con la funzione ***select()***.

È il caso di osservare che l'argomento che specifica il flusso è separato dalla lista di stringhe da emettere solo attraverso uno o più spazi (non si usa la virgola). Per lo stesso motivo, se il flusso di file è contenuto in un elemento di un array, oppure è il risultato di un'espressione, ciò deve essere indicato in un blocco.

Segue la descrizione di alcuni esempi.

- ```
  print MIOHANDLE "Ciao, come stai?\n";
  ```
  Scrive nel flusso di file indicato, a partire dalla posizione attuale del puntatore, il messaggio indicato come argomento.
- ```
  print {$elenco_file[$i]} "Bla bla bla\n";
  ```
  Inserisce il messaggio nel file indicato da '**`$elenco_file[$i]`**'.
- ```
  use Fcntl ':flock';   # importa le costanti LOCK_…
  …
  open (ELENCO, ">> /home/tizio/mioelenco");
  flock (ELENCO, LOCK_EX);
  print ELENCO $daelencare,"\n";
  flock (ELENCO, LOCK_UN);
  ```
  Vengono eseguite le operazioni seguenti:
  - si caricano le costanti di definizione dei tipi di blocco attraverso l'istruzione '**`use Fcntl ':flock';`**';
  - si apre il file '`/home/tizio/mioelenco`' in aggiunta;
  - si blocca il file in modo esclusivo;
  - si inserisce una riga nel file;
  - si rilascia il blocco.

24.16.3 Spostamento del puntatore

Lo spostamento del puntatore interno a un flusso di file avviene generalmente in modo automatico, sia in lettura, sia in scrittura. Si possono porre dei problemi, o dei dubbi, quando si accede simultaneamente a un file sia in lettura che in scrittura. Lo spostamento del puntatore può essere fatto attraverso la funzione ***seek()***.

```
seek flusso, posizione, partenza
```

La posizione effettiva nel file dipende dal valore del secondo e del terzo argomento. Precisamente, il terzo argomento può essere zero, uno o due, in base al significato seguente:

Partenza	Descrizione
0	la nuova posizione corrisponde esattamente a quanto indicato dal secondo argomento;
1	la nuova posizione corrisponde alla posizione corrente più quanto indicato nel secondo argomento;
2	la nuova posizione corrisponde alla posizione successiva alla fine del file più il valore del secondo argomento (solitamente negativo).

Segue la descrizione di alcuni esempi.

- ```
  seek (MIO_FILE, 0, 2);
  ```
  Posiziona alla fine del file in modo da poter, successivamente, aggiungere qualcosa a questo.
- ```
  seek (MIO_FILE, 0, 0);
  ```
  Posiziona all'inizio del file.
- ```
  use Fcntl ':flock';   # importa le costanti LOCK_…
  …
  open (ELENCO, ">> /home/tizio/mioelenco");
  flock (ELENCO, LOCK_EX);
  seek (ORDINI, 0, 2);
  print ELENCO $daelencare,"\n";
  flock (ELENCO, LOCK_UN);
  ```


Vengono eseguite le operazioni seguenti:

- si caricano le costanti di definizione dei tipi di blocco attraverso l'istruzione '**`use Fcntl ':flock';`**';
- si apre il file '`/home/tizio/mioelenco`' in aggiunta;
- si blocca il file in modo esclusivo;
- per sicurezza si posiziona il puntatore alla fine del file;
- si inserisce una riga nel file;
- si rilascia il blocco.

24.16.4 Identificazione dei flussi di file

Nel momento in cui si apre un file, si deve attribuire il nome del flusso relativo. Fino a questo punto è stato visto l'uso di nomi dichiarati nell'istante dell'apertura, come nell'esempio seguente:

```
open (MIO_FLUSSO, "< pippo.txt");
```

Da quel punto, il simbolo '**`MIO_FLUSSO`**' diviene ciò che identifica il flusso. È già stato mostrato anche il modo in cui è possibile trasferire il riferimento a questi simboli, come nell'esempio seguente:

```
$mio_flusso = \*MIO_FLUSSO;
```

Successivamente è possibile fare riferimento in modo indifferente al simbolo originale o alla variabile che vi punta:

```
$riga = <$mio_flusso>;
```

In realtà, il simbolo che rappresenta un flusso, può anche essere una variabile, contenente una stringa qualunque: è il contenuto della variabile che identifica effettivamente il flusso. Si osservi l'esempio seguente:

```
#!/usr/bin/perl
$a = "tizio";
open ($a, "< prova_1");
$a = "caio";
open ($a, "> prova_2");
$a = "tizio";
$riga = <$a>;
print STDOUT "$riga";
$riga = <"tizio">;
print STDOUT "$riga";
$a = "caio";
print $a "ciao\n";
print caio "come stai\n";
$a = "tizio";
close ($a);
$a = "caio";
close ($a);
```

Si vede la variabile '**`$a`**' che inizialmente riceve la stringa '**`tizio`**' e in questa situazione viene usata per aprire in lettura il file '`prova_1`'. Subito dopo, la stessa variabile riceve la stringa '**`caio`**' e in questo modo viene usata per aprire in scrittura il file '`prova_2`'. I due flussi sono identificati rispettivamente dalle stringhe '**`tizio`**' e '**`caio`**'; non ha importanza se queste stringhe sono contenute in una variabile o se sono usate direttamente come sono.

Più avanti, si può vedere che, quando '**`$a`**' contiene la stringa '**`tizio`**', scrivere

```
$riga = <$a>;
```

oppure

```
$riga = <"tizio">;
```

dà lo stesso risultato: la lettura del flusso abbinato al file '`prova_1`'. Ugualmente si può fare per il flusso in scrittura, con la differenza che non si può usare la stringa in modo delimitato:

```
$a = "caio";
print $a "ciao\n";
print caio "come stai\n";
```

Questa possibilità di gestire i flussi identificandoli subito attraverso delle variabili, facilita il trasferimento dell'indicazione dei flussi nelle chiamate di funzione, senza più il bisogno di creare dei riferimenti.

Si noti che non basta dichiarare un flusso indicando semplicemente una variabile, perché questa variabile deve essere inizializzata in qualche modo. Utilizzando una variabile non inizializzata sarebbe come volere identificare il flusso con la stringa nulla.

24.17 Funzioni interne

Nelle sezioni seguenti viene descritto brevemente il funzionamento di alcune funzioni interne di Perl. La sintassi viene mostrata secondo lo stile della documentazione di Perl, per cui, '***blocco***' rappresenta un gruppo di istruzioni nella forma consueta di Perl, e '***lista***' rappresenta un elenco di espressioni separate da virgole.

'***blocco***' equivale a:

```
{ istruzione... }
```

'***lista***' equivale a:

```
espressione1, espressione2, ...
```

Le funzioni descritte sono raggruppate in base al tipo di situazione in cui vengono utilizzate normalmente.

24.17.1 File

Vengono qui descritte alcune funzioni che riguardano la gestione dei file, nel senso globale, esterno. Le funzioni per la gestione del contenuto dei file vengono mostrate più avanti.

Tabella 24.156. Funzioni Perl per la gestione esterna dei file.

Funzione	Descrizione
`-x` ***nome_file*** `-x` ***flusso***	Perl permette di effettuare una serie di test sui file in modo analogo a quanto si fa con le shell tradizionali. Nel primo caso si fa riferimento a un file indicato per nome, nel secondo il riferimento è a un flusso di file. La lettera ***x*** cambia a seconda del tipo di test da verificare. I vari test restituiscono il valore uno se si verificano, oppure la stringa nulla in caso contrario (salvo eccezioni, come mostrato successivamente). Le descrizioni successive mostrano i vari tipi di test.
`-r` ***nome***	Il file è accessibile in lettura dal numero UID/GID efficace.
`-w` ***nome***	Il file è accessibile in scrittura dal numero UID/GID efficace.
`-x` ***nome***	Il file è accessibile in esecuzione dal numero UID/GID efficace.
`-o` ***nome***	Il file appartiene al numero UID efficace.
`-R` ***nome***	Il file è accessibile in lettura dal numero UID/GID reale.
`-W` ***nome***	Il file è accessibile in scrittura dal numero UID/GID reale.
`-X` ***nome***	Il file è accessibile in esecuzione dal numero UID/GID reale.
`-O` ***nome***	Il file appartiene al numero UID reale.
`-e` ***nome***	Il file esiste.
`-z` ***nome***	Il file ha dimensione zero.
`-s` ***nome***	Il file ha una dimensione maggiore di zero (restituisce la dimensione).
`-f` ***nome***	Si tratta di un file normale.
`-d` ***nome***	Si tratta di una directory.
`-l` ***nome***	Si tratta di un collegamento simbolico.
`-p` ***nome***	Si tratta di un file FIFO (*pipe* con nome).
`-S` ***nome***	Si tratta di un socket.

Funzione	Descrizione
`-b` ***nome***	Si tratta di file di dispositivo a blocchi.
`-c` ***nome***	Si tratta di file di dispositivo a caratteri.
`-t` ***nome***	Si tratta di un flusso di file aperto su un terminale.
`-u` ***nome***	Il file ha il bit SUID attivo.
`-g` ***nome***	Il file ha il bit SGID attivo.
`-k` ***nome***	Il file ha il bit Sticky attivo.
`-T` ***nome***	Si tratta di un file di testo.
`-B` ***nome***	Si tratta di un file binario.
`-M` ***nome***	Restituisce quanto tempo ha il file in base alla data di modifica.
`-A` ***nome***	Restituisce quanto tempo ha il file in base alla data di accesso.
`-C` ***nome***	Restituisce quanto tempo ha il file in base alla data di creazione.
`chmod` ***permessi, file, ...***	***chmod()*** cambia i permessi dei file indicati come argomento. In particolare, l'argomento è una lista, in cui il primo elemento è costituito dai permessi espressi in forma numerica ottale. Dal momento che si tratta di un numero ottale, è bene che non sia fornito in forma di stringa perché la conversione da stringa a numero ottale non è automatica. Restituisce il numero di file su cui ha potuto intervenire con successo.
`chown` ***uid, gid, file, ...***	***chown()*** cambia i permessi dei file indicati nella lista di argomenti. I primi due elementi della lista sono rispettivamente il numero UID e GID. Gli elementi restanti sono i file su cui si vuole intervenire. Restituisce il numero di file su cui ha potuto intervenire con successo.
`link` ***file_di_origine,*** ↩ ↪***collegamento_di_destinazione***	***link()*** genera un collegamento fisico a partire da un file esistente. Restituisce *Vero* se la creazione ha successo.
`lstat` ***file*** `lstat` ***flusso***	***lstat()*** funziona esattamente come ***stat()***, con la differenza che restituisce le informazioni relative a un collegamento simbolico, invece di quelle del file a cui questo punta. Se non viene indicato l'argomento, ***lstat()*** utilizza il contenuto della variabile predefinite '**$_**'.
`readlink` ***file***	***readlink()*** restituisce il valore di un collegamento simbolico. Se non viene indicato l'argomento, ***readlink()*** utilizza il contenuto della variabile predefinita '**$_**'.
`rename` ***nome_vecchio,*** ↩ ↪***nome_nuovo***	***rename()*** cambia il nome di un file, o lo sposta. Tuttavia, lo spostamento non può avvenire al di fuori del file system di partenza. Restituisce uno se l'operazione riesce, altrimenti zero.

Funzione	Descrizione
`stat` ***file*** `stat` ***flusso***	***stat()*** restituisce un array di tredici elementi contenenti tutte le informazioni sul file indicato per nome o attraverso un flusso di file. Se non viene indicato l'argomento, ***stat()*** utilizza il contenuto della variabile predefinita '**$_**'. Gli elementi dell'array restituito sono riportati nella tabella 24.157 in cui appare anche il nome suggerito per la trasformazione in variabili scalari. Va osservato che le informazioni data-orario sui file sono espresse in forma numerica che esprime il tempo trascorso a partire dalla data di riferimento del sistema operativo. Nel caso dei sistemi derivati da Unix si tratta dell'ora zero del 1/1/1970. Nello stesso modo, è evidente che tutte queste informazioni possono essere ottenute solo da un file system che può gestirle.
`symlink` ***file_di_origine*** `,` ↩ ↪***collegamento_di_destinazione***	***symlink()*** genera un collegamento simbolico a partire da un file esistente. Restituisce *Vero* se la creazione ha successo.
`unlink` ***lista_di_file***	***unlink()*** cancella i file indicati per nome tra gli argomenti. Generalmente non possono essere cancellate le directory (e comunque sarebbe inopportuno dato il tipo di cancellazione che si fa). Restituisce il numero di file cancellati con successo. Se non viene indicato l'argomento, ***unlink()*** utilizza il contenuto della variabile predefinita '**$_**'.
`utime` ***data_di_accesso*** `,` ↩ ↪***data_di_modifica*** `,` ***lista_di_file***	***utime()*** cambia la data di modifica e di accesso di una serie di file. Le date, indicate come argomenti iniziali, sono espresse nella forma numerica gestita dal sistema operativo. La data di modifica dell'inode viene cambiata automaticamente in modo che corrisponda al momento in cui questa modifica viene effettuata.

Tabella 24.157. Elenco degli elementi componenti l'array restituito da ***stat()***.

Elemento	Nome consueto	Descrizione.
0	`$dev`	Numero del dispositivo del file system.
1	`$ino`	Numero dell'inode.
2	`$mode`	Permessi del file.
3	`$nlink`	Numero di collegamenti fisici al file.
4	`$uid`	UID dell'utente proprietario del file.
5	`$gid`	GID del gruppo proprietario del file.
6	`$rdev`	Identificatore di dispositivo, per i file speciali.
7	`$size`	Dimensione in byte.
8	`$atime`	Data dell'ultimo accesso.
9	`$mtime`	Data dell'ultima modifica.
10	`$ctime`	Data di cambiamento di inode.
11	`$blksize`	Dimensione preferita dei blocchi per le operazioni di I/O del sistema.
12	`$blocks`	Numero di blocchi allocati attualmente.

Segue la descrizione di alcuni esempi.

- ```
  if (-x "esempio.pl")
    {
      print "Il file è eseguibile\n";
    }
  ```
  Restituisce il messaggio se il file '`esempio.pl`' è eseguibile.
- ```
  chmod 0755, 'mio_file', 'tuo_file', 'suo_file';
  ```
  Cambia i permessi ai file indicati dopo la modalità.
- ```
  @elenco = ('mio_file', 'tuo_file', 'suo_file');
  chmod 0755, @elenco;
  ```
  Esattamente come nell'esempio precedente.
- ```
  @elenco = ('mio_file', 'tuo_file', 'suo_file');
  chmod (0755, @elenco);
  ```
  Esattamente come nell'esempio precedente, ma più simile alle chiamate di funzione degli altri linguaggi.
- ```
  chown 1001, 100, 'mio_file', 'tuo_file', 'suo_file';
  ```
  Cambia l'utente e il gruppo proprietari dei file '`mio_file`', '`tuo_file`' e '`suo_file`'.
- ```
  chown (1001, 100, 'mio_file', 'tuo_file', 'suo_file');
  ```
  Esattamente come nell'esempio precedente.
- ```
  $prova = readlink '/bin/sh';
  ```
  Assegna alla variabile '**$prova**' il percorso contenuto nel collegamento simbolico '`/bin/sh`'. Probabilmente, alla fine, la variabile contiene la stringa '**bash**'.
- ```
  ($dev, $ino, $mode, $nlink,
   $uid, $gid, $rdev, $size,
   $atime, $mtime, $ctime,
   $blksize, $blocks) = stat ('/home/tizio/mio_file');
  ```
  Preleva tutte le informazioni sul file '`/home/tizio/mio_file`' e le scompone in diverse variabili scalari.
- ```
  $momento = time;
  utime $momento, $momento, 'mio_file';
  ```
  Cambia la data di accesso e modifica in modo da farle coincidere con quella riportata dall'orologio dell'elaboratore nel momento in cui si eseguono queste istruzioni.

24.17.2 Directory

Vengono qui elencate alcune funzioni che riguardano la gestione delle directory e di raggruppamenti di file. Vengono ignorate volutamente le funzioni specifiche di Perl per la lettura delle directory.

Tabella 24.167. Funzioni Perl per la gestione delle directory e dei raggruppamenti di file.

Funzione	Descrizione
`chdir` ***directory***	***chdir()*** cambia la directory di lavoro posizionandosi in corrispondenza di quanto indicato come argomento. Se l'argomento viene omesso, lo spostamento avviene nella directory personale, attraverso quanto determinato dal contenuto di '**$ENV{"HOME"}**'. Restituisce *Vero* se l'operazione ha successo, *Falso* in tutti gli altri casi.
`glob` ***espressione***	***glob()*** restituisce quanto indicato nell'argomento dopo un'operazione di espansione, come farebbe una shell. Se l'argomento non viene indicato, l'espansione viene effettuata sul contenuto della variabile '**$_**'.
  ```

Funzione	Descrizione
`mkdir` ***directory***`,` ***permessi***	***mkdir()*** crea la directory indicata come primo argomento. I permessi della directory sono indicati come secondo argomento, devono essere espressi con un numero ottale, tenendo conto che poi vengono filtrati ulteriormente dalla maschera dei permessi. Restituisce uno se l'operazione riesce, altrimenti zero, impostando anche la variabile '**`$!`**'. In generale, non dovrebbe essere possibile assegnare dei permessi negli S-bit. In pratica dovrebbe essere consentito di operare solo con i soliti permessi di lettura, scrittura ed esecuzione (attraversamento).
`rmdir` ***directory***	***rmdir()*** elimina la directory indicata come argomento. Se l'argomento non viene fornito, si utilizza la variabile predefinita '**`$_`**'. Restituisce uno se l'operazione riesce, altrimenti zero, impostando anche la variabile '**`$!`**'.

Segue la descrizione di alcuni esempi.

- `$primo = glob ("/bin/*");`
  Assegna alla variabile '**`$primo`**' il percorso assoluto del primo file che viene trovato attraverso l'espansione del modello '`/bin/*`'.
- `@elenco = glob ("/bin/*");`
  Assegna all'array '**`@elenco`**' i percorsi assoluti dei file che vengono trovati attraverso l'espansione del modello '`/bin/*`'.
- `mkdir ("/tmp/prova");`
  Crea la directory '`/tmp/prova/`' con i permessi normali dell'utente.[1]
- `mkdir ("/tmp/prova", 0755);`
  Crea la directory '`/tmp/prova/`' con i permessi 0755_8 (si osservi che si tratta di un numero ottale), che vengono comunque filtrati dalla maschera dei permessi.

24.17.3 I/O

Vengono elencate alcune funzioni che riguardano la gestione dei dati contenuti nei file.

Tabella 24.172. Funzioni Perl per la gestione del contenuto dei file.

Funzione	Descrizione
`binmode` ***flusso***`, ":`***codifica***`"`	***binmode()*** consente di dichiarare la codifica utilizzata per il file corrispondente al flusso di file indicato come primo argomento. Si usa normalmente quando si vuole dichiarare la codifica dei flussi standard, che risultano già aperti senza alcuna dichiarazione esplicita.
`chomp` ***espressione_stringa*** `chomp` ***lista***	***chomp()*** riceve come argomento un'espressione che restituisce una stringa o una lista di stringhe. Il suo scopo è eliminare dalla parte finale il codice di interruzione di riga, che coincide normalmente con il carattere *<LF>*. Precisamente si tratta di quanto contenuto nella variabile predefinita '**`$/`**'. Se non viene indicato l'argomento, interviene sul contenuto della variabile '**`$_`**'. Restituisce il numero di caratteri eliminati.

Funzione	Descrizione
`chop` ***espressione_stringa*** `chop` ***lista***	***chop()*** riceve come argomento un'espressione che restituisce una stringa o una lista di stringhe. Il suo scopo è eliminare l'ultimo carattere della stringa, o delle stringhe della lista. In questo senso differisce da ***chomp()*** che invece elimina la parte finale solo se necessario. Restituisce l'ultimo carattere eliminato.
`close` ***flusso***	***close()*** chiude un flusso di file aperto precedentemente. Restituisce *Vero* se l'operazione ha successo e non si sono prodotti errori di alcun tipo. È opportuno osservare che non è necessario chiudere un file se poi si deve riaprire immediatamente con la funzione ***open()***: lo si può semplicemente riaprire.
`eof` ***flusso***	***eof()*** verifica se la prossima lettura del flusso di file supera la fine del file. Restituisce uno se ciò si verifica. Questa funzione è generalmente di scarsa utilità dal momento che la lettura di una riga oltre la fine del file genera un risultato indefinito che può essere verificato tranquillamente in un'espressione condizionale. Oltre a ciò, ***eof()*** si verifica prima che il tentativo di lettura sia stato fatto veramente, contrariamente a quanto avviene di solito in altri linguaggi di programmazione.
`fcntl` ***flusso***`,` ***funzione***`,` ***scalare***	***fcntl()*** esegue la chiamata di sistema omonima e per questo può essere utilizzata solo con un sistema operativo che la gestisce. Prima di poter utilizzare questa funzione occorre richiamare una serie di valori corrispondenti a macro del proprio sistema: '**`use Fcntl;`**'
`fileno` ***flusso***	***fileno()*** restituisce il descrittore corrispondente a un flusso di file.
`flock` ***flusso***`,` ***operazione***	***flock()*** esegue la chiamata di sistema omonima, oppure una sua emulazione, per il file identificato tramite il flusso di file. ***flock()*** permette di eseguire il blocco di un file nel suo complesso e non record per record. Restituisce *Vero* se l'operazione ha successo. L'operazione, cioè il tipo di blocco, viene indicata attraverso una sorta di macro che viene inserita nel sorgente di Perl attraverso la dichiarazione seguente: '**`use Fcntl ':flock';`**'
`flock` ***flusso***`, LOCK_SH`	'**`LOCK_SH`**' corrisponde normalmente al valore numerico uno. Richiede un blocco condiviso (*shared*).
`flock` ***flusso***`, LOCK_EX`	'**`LOCK_EX`**' corrisponde normalmente al valore numerico due. Richiede un blocco esclusivo.
`flock` ***flusso***`, LOCK_UN`	'**`LOCK_UN`**' corrisponde normalmente al valore numerico otto. Rilascia il blocco.

Funzione	Descrizione
`flock` ***flusso***`, LOCK_NB`	'**`LOCK_NB`**' corrisponde normalmente al valore numerico quattro. Viene sommato a '**`LOCK_SH`**' o a '**`LOCK_EX`**' in modo da non attendere lo sblocco del file nel caso che questo risulti già bloccato.
`getc` ***flusso***	***getc()*** legge il file indicato dal flusso di file, o dallo standard input se viene omesso l'argomento, restituendo il prossimo carattere. Se si supera la fine del file restituisce la stringa nulla.
`ioctl` ***flusso***`,` ***funzione***`,` ***scalare***	***ioctl()*** esegue la chiamata di sistema omonima e per questo può essere utilizzata solo con un sistema operativo che la gestisce. Per poterla utilizzare occorre consultare la documentazione interna di Perl.
`open` ***flusso***`,` `[` ***modalità***`,` `]` ***file***	***open()*** apre il file indicato come ultimo argomento utilizzando il flusso di file indicato come primo argomento. Se manca l'argomento centrale, il nome del file è composto normalmente da un prefisso simbolico che ne rappresenta la modalità di utilizzo. Il prefisso può essere staccato dal nome del file attraverso spazi. L'apertura del file rappresentato da un trattino ('`-`') è equivalente all'apertura dello standard input, mentre l'apertura del file '`>-`' è equivalente all'apertura dello standard output. Restituisce *Vero* se l'apertura ha successo.
`open` ***flusso***`, "<`***file***`"` `open` ***flusso***`, "<:`***codifica***`", "`***file***`"`	Questo simbolo o l'assenza di ogni altro prefisso rappresenta l'apertura del file in lettura, o in input.
`open` ***flusso***`, ">`***file***`"` `open` ***flusso***`, ">:`***codifica***`", "`***file***`"`	Il file viene troncato (viene ridotto a un file vuoto) e aperto in scrittura, o in output.
`open` ***flusso***`, ">>`***file***`"` `open` ***flusso***`, ">>:`***codifica***`", "`***file***`"`	Il file viene aperto in scrittura in aggiunta.
`open` ***flusso***`, "+<`***file***`"` `open` ***flusso***`, "+<:`***codifica***`", "`***file***`"`	Il file viene aperto in lettura e scrittura, senza il troncamento iniziale.
`open` ***flusso***`, "+>`***file***`"` `open` ***flusso***`, "+>:`***codifica***`", "`***file***`"`	Il file viene aperto in scrittura e lettura, a cominciare dal troncamento iniziale.
`open` ***flusso***`, "+>>`***file***`"` `open` ***flusso***`, "+>>:`***codifica***`", "`***file***`"`	Il file viene aperto in aggiunta e in lettura.
`open` ***flusso***`, "|`***comando***`"`	Il file viene interpretato come un comando a cui inviare i dati in scrittura attraverso un condotto.
`open` ***flusso***`, "`***comando***`|"`	Il file viene interpretato come un comando da cui leggere i dati emessi dal suo standard output.
`open` ***flusso***`, "|`***comando***`|"`	Il file viene interpretato come un comando a cui inviare i dati in scrittura e attraverso il suo standard input, leggendo quanto emesso attraverso lo standard output.

Funzione	Descrizione
`pipe` ***flusso_in_lettura***`,` ↩ ↪***flusso_in_scrittura***	***pipe()*** esegue la chiamata di sistema omonima, aprendo due flussi di file, uno in lettura e l'altro in scrittura. Per poterla utilizzare occorre consultare la documentazione interna di Perl.
`print` ***flusso lista*** `print` ***lista***	***print()*** emette attraverso il flusso di file indicato la lista di argomenti successiva. Se non viene specificato un flusso di file, tutto viene emesso attraverso lo standard output, oppure attraverso quanto specificato con la funzione ***select()***. Se non viene specificato alcun argomento, viene emesso il contenuto della variabile '**`$_`**'. È il caso di osservare che l'argomento che specifica il flusso di file è separato dalla lista di stringhe da emettere solo attraverso uno o più spazi (non si usa la virgola). Per lo stesso motivo, se il flusso di file è contenuto in un elemento di un array, oppure è il risultato di un'espressione, ciò deve essere indicato in un blocco. Restituisce *Vero* se l'operazione di scrittura ha successo.
`printf` ***flusso formato***`,` ***lista*** `printf` ***formato***`,` ***lista***	È equivalente all'uso di ***sprintf()*** nel modo seguente: '**`print`** ***flusso*** **`sprintf`** ***formato***`,` ***lista***'
`read` ***flusso***`,` ***scalare***`,` ***lunghezza***`,` ↩ ↪***scostamento*** `read` ***flusso***`,` ***scalare***`,` ***lunghezza***	***read()*** tenta di leggere il flusso di file specificato e di ottenere la quantità di byte espressa nel terzo argomento, inserendo quanto letto nella variabile scalare indicata come secondo. Se viene indicato anche il quarto argomento, lo scostamento, il contenuto della variabile non viene rimpiazzato completamente, ma è sovrascritto a partire dalla posizione indicata dallo scostamento stesso. La funzione restituisce il numero di byte letti effettivamente, oppure il valore indefinito se si è verificato un errore.
`seek` ***flusso***`,` ***posizione***`,` ***partenza***	***seek()*** modifica la posizione del puntatore riferito al flusso di file. La posizione effettiva nel file dipende dal valore del secondo e del terzo argomento. Precisamente, il terzo argomento può essere zero, uno o due, come descritto nelle voci successive.
`seek` ***flusso***`,` ***posizione***`, SEEK_SET` `seek` ***flusso***`,` ***posizione***`, 0`	Sposta il puntatore esattamente a quanto indicato dal secondo argomento.
`seek` ***flusso***`,` ***posizione***`, SEEK_CUR` `seek` ***flusso***`,` ***posizione***`, 1`	Sposta il puntatore alla posizione corrente, più quanto indicato dal secondo argomento.
`seek` ***flusso***`,` ***posizione***`, SEEK_END` `seek` ***flusso***`,` ***posizione***`, 2`	Sposta il puntatore alla fine del file, più quanto indicato dal secondo argomento, che solitamente è un valore negativo.
`select` ***flusso***	***select()*** permette di definire il flusso di file in scrittura predefinito, per tutte quelle situazioni in cui questo concetto ha significato.

Funzione	Descrizione
`sprintf` ***formato*** **,** ***lista***	***sprintf()*** restituisce una stringa formattata in modo analogo a quanto fa la funzione omonima del linguaggio C. Il primo argomento è la stringa di composizione, quelli successivi sono i valori da inserire. Perl utilizza una propria gestione della conversione secondo quanto riportato nelle tabelle 24.173 e 24.174. ***sprintf()*** è sensibile all'attivazione della localizzazione, nel qual caso, il carattere utilizzato per separare le cifre intere da quelle decimali, dipende dalla variabile di ambiente '**`LC_NUMERIC`**'.
`tell` ***flusso***	***tell()*** restituisce la posizione corrente del puntatore interno riferito al flusso di file indicato come argomento, oppure a quello dell'ultima operazione di lettura eseguita.
`use open ":`***codifica***`"`	'**`use open`**' consente di dichiarare la codifica una volta per tutte, fino a quando si incontra un'altra istruzione del genere.

Tabella 24.173. Elenco dei simboli utilizzabili in una stringa formattata per l'utilizzo con ***sprintf()***.

Simbolo	Corrispondenza
`%%`	Segno di percentuale.
`%c`	Un carattere con il numero dato.
`%s`	Una stringa.
`%d`	Un intero con segno a base 10.
`%u`	Un intero senza segno a base 10.
`%o`	Un intero senza segno in ottale.
`%x`	Un intero senza segno in esadecimale.
`%e`	Un numero a virgola mobile, in notazione scientifica.
`%f`	Un numero a virgola mobile, in notazione decimale fissa.
`%g`	Un numero a virgola mobile, secondo la notazione di '**`%e`**' o '**`%f`**'.
`%X`	Come '**`%x`**', ma con l'uso di lettere maiuscole.
`%E`	Come '**`%e`**', ma con l'uso della lettera '**`E`**' maiuscola.
`%G`	Come '**`%g`**', ma con l'uso della lettera '**`E`**' maiuscola (se applicabile).
`%p`	Un puntatore (l'indirizzo utilizzato da Perl in esadecimale).
`%n`	Immagazzina, nella prossima variabile, il numero di caratteri già emessi.
`%i`	Sinonimo di '**`%d`**'.
`%D`	Sinonimo di '**`%ld`**'.
`%U`	Sinonimo di '**`%lu`**'.
`%O`	Sinonimo di '**`%lo`**'.
`%F`	Sinonimo di '**`%f`**'.

Tabella 24.174. Elenco dei simboli utilizzabili tra il segno di percentuale e la lettera di conversione.

Simbolo	Corrispondenza
`spazio`	Il prefisso di un numero positivo è uno spazio.

Simbolo	Corrispondenza
`+`	Il prefisso di un numero positivo è il segno '**`+`**'.
`-`	Allinea a sinistra rispetto al campo.
`0`	Utilizza zeri, invece di spazi, per allineare a destra.
`#`	Prefissa un numero ottale con uno zero e un numero esadecimale con 0x...
n	Un numero definisce la dimensione minima del campo.
`.`***n***	Per i numeri a virgola mobile esprime la precisione, ovvero il numero di decimali.
`.`***n***	Per le stringhe definisce la lunghezza massima.
`.`***n***	Per gli interi definisce la lunghezza minima.
`l`	Interpreta un intero come il tipo C '**`long`**' o '**`unsigned long`**'.
`h`	Interpreta un intero come il tipo C '**`short`**' o '**`unsigned short`**'.
`V`	Interpreta un intero secondo il tipo standard di Perl.

Segue la descrizione di alcuni esempi.

- ```
  #!/usr/bin/perl
  $/ = "\r\n";
  while ($riga = <STDIN>)
    {
      chomp ($riga);
      print STDOUT ("$riga\n");
    }
  ```
  Quello che si vede è un esempio molto semplice di un filtro che trasforma un file di testo in stile Dos a uno in stile Unix. In pratica, viene definito che l'interruzione di riga è indicata attraverso la sequenza dei caratteri <*CR*><*LF*> ('**`\r\n`**'), che attraverso la funzione ***chomp()*** viene eliminata dalle righe lette. Infine, le righe vengono emesse attraverso lo standard output, con l'aggiunta del codice <*LF*> finale.

- ```
  close (MIO_FILE);
  ```
  Chiude il flusso di file '**`MIO_FILE`**'.

- ```
  use Fcntl ':flock';   # importa le costanti LOCK_…
  …
  open (ELENCO, ">>/home/tizio/mioelenco");
  flock (ELENCO, LOCK_EX);
  seek (ELENCO, 0, 2);
  print ELENCO $daelencare,"\n";
  flock (ELENCO, LOCK_UN);
  ```
  Vengono eseguite le operazioni seguenti:
  - si caricano le costanti di definizione dei tipi di blocco attraverso l'istruzione '**`use Fcntl ':flock';`**';
  - si apre il file '`/home/tizio/mioelenco`' in aggiunta;
  - si blocca il file in modo esclusivo;
  - per sicurezza si posiziona il puntatore del file alla fine dello stesso;
  - si inserisce una riga nel file;
  - si rilascia il blocco.

- ```
  use Fcntl ':flock';   # importa le costanti LOCK_…
  …
  open (ELENCO, ">> /home/tizio/mioelenco");
  if (flock (ELENCO, (LOCK_EX)+(LOCK_NB)))
    {
      seek (ELENCO, 0, 2);
      print ELENCO $daelencare,"\n";
      flock (ELENCO, LOCK_UN);
    }
  else
    {
      print STDOUT "Il file è impegnato.\n";
    }
  ```
  Si tratta di una variante dell'esempio precedente in cui si richiede un blocco esclusivo senza attesa. Se il blocco ha successo, si procede, altrimenti viene segnalata la presenza del blocco eseguito

da un altro processo (si osservi il fatto che le macro sono state racchiuse tra parentesi tonde prima di sommarle assieme).

```
if (open (ORDINI, ">> /var/log/ordini"))
  {
    if (flock (ORDINI, LOCK_EX))
      {
        seek (ORDINI, 0, 2);
        print ORDINI ("$ordine\n");
      }
    close (ORDINI);
  }
```

Tenta di aprire il file '`/var/log/ordini`' in aggiunta, quindi tenta di bloccarlo in modo esclusivo. Se ci riesce sposta il puntatore alla fine del file, per sicurezza, quindi inserisce un nuovo ordine. Infine chiude il file.

```
if (open (MAN, "man $DATI{sezione} $DATI{man} | col -bx |"))
  {
    print "Content-type: text/html\n";
    print "\n";
    print "<HTML>\n";
    print "<HEAD>\n";
    print "<TITLE>man $DATI{sezione} $DATI{man}</TITLE>\n";
    print "</HEAD>\n";
    print "<BODY>\n";
    print "<H1>man $DATI{sezione} $DATI{man}</H1>\n";
    print "<PRE>\n";

    while ($risposta = <MAN>)
      {
        print $risposta;
      }

    print "</PRE>\n";
    print "</BODY>\n";
    print "</HTML>\n";
}
```

Genera una pagina HTML a partire da un comando '**man**'.

- `print "Ciao, come stai?\n";`

  Emette attraverso lo standard output il messaggio indicato come argomento.

- `print STDERR "Errore $errore\n";`

  Emette attraverso lo standard error il messaggio indicato come argomento.

- `print { $elenco_file[$i] } "Bla bla bla\n";`

  Inserisce il messaggio nel flusso di file indicato da '**$elenco_file[$i]**'.

- `print { $ok ? STDOUT : STDERR } ("Bla bla bla\n");`

  Emette il messaggio attraverso lo standard output, oppure lo standard error, a seconda del valore contenuto in '**$ok**'.

- `seek (MIO_FILE, 0, 2);`

  Posiziona alla fine del file in modo da poter aggiungere successivamente qualcosa a questo.

- `seek (MIO_FILE, 0, 0);`

  Posiziona all'inizio del file.

```
select (MIO_FILE);
…
print ("ciao a tutti\n");
```

Aggiunge al file identificato dal flusso di file '**MIO_FILE**' il messaggio '**ciao a tutti**'.

24.17.4 Interazione con il sistema

Vengono descritte alcune funzioni per l'interazione con il sistema.

Tabella 24.188. Funzioni Perl per l'interazione con il sistema operativo.

Funzione	Descrizione
`exec` ***elenco***	***exec()*** avvia l'esecuzione del comando indicato negli argomenti, senza riprendere l'esecuzione del programma al termine. Si comporta quindi in modo analogo al comando interno omonimo delle shell comuni.
`kill` ***segnale*** `,` ***elenco_di_processi***	***kill()*** invia un segnale a una serie di processi. Il primo argomento deve essere il segnale. Restituisce il numero di processi che hanno ricevuto il segnale.
`sleep` ***secondi***	***sleep()*** mette in pausa l'esecuzione del programma, per il numero di secondi indicato come argomento, eventualmente attraverso un'espressione. Se l'argomento non viene indicato, la pausa non ha fine. L'attesa può essere interrotta inviando un segnale '**SIGALRM**' al processo. Restituisce il numero di secondi trascorsi effettivamente.
`system` ***elenco***	***system()*** avvia l'esecuzione del comando indicato negli argomenti, attende la sua conclusione e restituisce il valore generato dal comando stesso.
`time`	***time()*** restituisce la data e l'ora attuale espressa in secondi trascorsi dalla data iniziale gestita dal sistema. Nel caso della maggior parte dei sistemi Unix si tratta dell'ora zero del 1/1/1970. Il valore ottenuto da ***time()*** può essere utilizzato dalle funzioni ***gmtime()*** e ***localtime()***.
`times`	***times()*** restituisce un array di quattro elementi che indicano rispettivamente: orario dell'utente; orario di sistema; orario dell'utente del processo figlio; orario di sistema del processo figlio.
`umask` ***maschera_numerica***	***umask()*** permette di definire la maschera dei permessi per il processo elaborativo del programma. Restituisce il valore precedente. La maschera è espressa in forma numerica; ciò significa che se la maschera da indicare come argomento è una stringa, potrebbe essere necessario l'utilizzo della funzione ***oct()*** per garantire l'interpretazione ottale e non a base 10.

Segue la descrizione di alcuni esempi.

- `exec ("ls");`

  Esegue il comando '**ls**' e conclude il funzionamento del programma. In pratica, le istruzioni successive a ***exec()***, non vengono eseguite.

- `kill ("TERM", 588);`

  Invia il segnale '**SIGTERM**' al processo numero 588.

- `kill (15, 588);`

  Esattamente come nell'esempio precedente.

- `kill (-15, 588);`

  Come nell'esempio precedente, ma il segnale viene inviato anche a tutti i processi discendenti da quello indicato.

- `sleep;`

  Mette il programma in pausa senza specificare la fine di questa.

- `sleep (10);`

  Mette il programma in pausa per 10 secondi.

- `sleep ($pausa);`

  Mette il programma in pausa per la quantità di secondi indicata dalla variabile '**$pausa**'.

- `system ("ls");`

  Esegue il comando '**ls**' e poi riprende con il programma.

```
if (system ("mkdir ciao")
  {
    die("La creazione della directory è fallita\n");
  }
else
  {
    print ("La directory è stata creata\n");
  }
```

L'esempio mostra il caso in cui si voglia controllare l'esito di un comando di sistema avviato attraverso la funzione ***system()***. Se il comando '**`mkdir ciao`**' viene eseguito con successo, restituisce il valore zero, che per Perl equivale a *Falso*. Quindi, se la condizione si avvera, significa che l'operazione è fallita, altrimenti, tutto è andato bene.

-
```
($user, $system, $cuser, $csystem) = times;
```
Scompone l'array restituito da ***times()*** in quattro variabili scalari.

-
```
$maschera = '644';
umask (oct ($maschera));
```
Modifica la maschera dei permessi in modo che sia pari a 0644_8. Dal momento che l'informazione è contenuta in una stringa, che per di più non ha lo zero iniziale della rappresentazione ottale convenzionale, occorre convertire prima la stringa in numero nel modo corretto.

24.17.5 Funzioni matematiche

Perl fornisce una serie di funzioni matematiche tipiche della maggior parte dei linguaggi di programmazione.

Tabella 24.200. Funzioni Perl di tipo matematico.

Funzione	Descrizione
`abs` ***x***	***abs()*** restituisce il valore assoluto del suo argomento. Se l'argomento non viene indicato, si utilizza la variabile predefinita '**`$_`**'.
`atan2` ***x,y***	***atan2()*** restituisce l'arcotangente nell'intervallo da −PI a +PI.
`cos` ***x***	***cos()*** restituisce il coseno. Se l'argomento non viene indicato, si utilizza la variabile predefinita '**`$_`**'.
`exp` ***x***	***exp()*** restituisce ***e*** (la base del logaritmo naturale) elevato al valore di ***x***, cioè dell'argomento. Se l'argomento non viene indicato, si utilizza la variabile predefinita '**`$_`**'.
`int` ***x***	***int()*** restituisce la parte intera del numero (o dell'espressione) fornito come argomento. Se l'argomento non viene indicato, si utilizza la variabile predefinita '**`$_`**'.
`log` ***x***	***log()*** restituisce il logaritmo naturale del valore fornito come argomento. Se l'argomento non viene indicato, si utilizza la variabile predefinita '**`$_`**'.
`sin` ***x***	***sin()*** restituisce il seno. Se l'argomento non viene indicato, si utilizza la variabile predefinita '**`$_`**'.
`sqrt` ***x***	***sqrt()*** restituisce la radice quadrata. Se l'argomento non viene indicato, si utilizza la variabile predefinita '**`$_`**'.

24.17.6 Funzioni di conversione

Nel seguito sono elencate le funzioni che si occupano di convertire dati in formati differenti.

Tabella 24.201. Funzioni Perl per la conversione dei dati.

Funzione	Descrizione
`chr` ***n***	***chr()*** restituisce il carattere corrispondente al numero indicato come argomento. Se non viene specificato l'argomento, il numero viene letto dalla variabile '**`$_`**'.
`hex` ***stringa***	***hex()*** interpreta il proprio argomento come una stringa contenente un numero esadecimale. Restituisce il numero (decimale) corrispondente. Se non viene specificato l'argomento, il dato viene letto dalla variabile '**`$_`**'.
`oct` ***stringa***	***oct()*** interpreta il proprio argomento come una stringa contenente un numero ottale. Restituisce il numero (decimale) corrispondente. Se non viene specificato l'argomento, il dato viene letto dalla variabile '**`$_`**'.
`ord` ***stringa***	***ord()*** restituisce il valore numerico corrispondente al codice ASCII del primo carattere della stringa fornita come argomento. Se non viene specificato l'argomento, il dato viene letto dalla variabile '**`$_`**'.

Segue la descrizione di alcuni esempi.

-
```
chr (65);
```
Restituisce la lettera '**`A`**' maiuscola.

-
```
hex ("0xAf");
```
Restituisce il numero 175.

-
```
hex ("af");
```
Restituisce il numero 175.

-
```
$permessi = '0755';
mkdir ("/tmp/prova", oct ($permessi));
```
Crea la directory '`/tmp/prova/`' con i permessi 0755_8. Dal momento che questi permessi sono contenuti in una variabile, in forma di stringa, devono essere convertiti in ottale prima dell'uso, altrimenti verrebbero interpretati in forma decimale.

24.17.7 Gestione delle espressioni

Sono elencate nel seguito le funzioni che si occupano di gestire l'esecuzione delle espressioni (quando necessario) e di conoscerne alcune caratteristiche.

Tabella 24.206. Funzioni Perl per il controllo delle espressioni e delle loro caratteristiche.

Funzione	Descrizione
`defined` ***espressione***	***defined()*** restituisce *Vero* se l'espressione (o la variabile) restituisce un valore diverso da indefinito. Il valore indefinito può essere restituito in particolare nelle situazioni seguenti: la lettura oltre la fine del file; un errore di sistema; una variabile non ancora inizializzata. È importante non confondere il valore indefinito con lo zero o la stringa nulla: si tratta di tre cose differenti, in particolare, zero e stringa nulla sono valori definiti.
`scalar` ***espressione***	***scalar()*** restituisce il risultato dell'espressione valutato in un contesto espressamente scalare.

24.17.8 Array e hash

Vengono elencate qui le funzioni che sono particolarmente dedicate alla gestione di array e hash.

Tabella 24.207. Funzioni Perl per la gestione di array.

Funzione	Descrizione
`delete` ***espressione***	***delete()*** elimina uno o più elementi da un hash. L'espressione che rappresenta l'argomento della funzione deve rappresentare uno o più elementi dell'hash. Restituisce i valori cancellati, cioè quelli abbinati alle chiavi indicate per la cancellazione.
`exists` ***espressione***	***exists()*** verifica l'esistenza di una chiave all'interno di un hash. Se esiste, anche se il valore corrispondente dovesse risultare indefinito, restituisce *Vero*. L'espressione che rappresenta l'argomento della funzione deve rappresentare un solo elemento dell'hash.
`keys` ***hash***	***keys()*** restituisce un array composto da tutte le chiavi dell'hash posto come argomento.
`pop` ***array***	***pop()*** restituisce l'ultimo elemento dell'array eliminandolo dall'array stesso (accorciandolo). In pratica tratta l'array come una pila (*stack*) ed esegue un'azione di *pop*.
`push` ***array*****,** ***lista***	***push()*** aggiunge all'array indicato come primo argomento gli elementi della lista successiva. In pratica tratta l'array come una pila (*stack*) ed esegue un'azione di *push*.
`splice` ***array*****,** ***posizione_iniziale*****,** ***lunghezza*****,** ***lista*** `splice` ***array*****,** ***posizione_iniziale*****,** ***lunghezza*** `splice` ***array*****,** ***posizione_iniziale***	***splice()*** elimina dall'array, indicato come primo argomento, gli elementi collocati a partire dalla posizione iniziale, indicata come secondo argomento, per una quantità definita dal terzo argomento. Se il terzo argomento (la quantità di elementi da eliminare) viene omesso, vengono eliminati tutti gli elementi a partire dalla posizione iniziale. Se dopo il numero di argomenti da eliminare appaiono altri argomenti, vengono interpretati come una lista da inserire in sostituzione degli elementi cancellati. In tal modo, attraverso questa funzione, si può accorciare e allungare un array a piacimento, intervenendo in qualunque punto dello stesso.

Segue la descrizione di alcuni esempi.

- ```
  delete $miohash{ $miachiave };
  ```
  Elimina dall'hash '**`%miohash`**' l'elemento rappresentato dalla chiave contenuta nella variabile '**`$miachiave`**'.

- ```
  if (exists $miohash{ $miachiave })
    {
      …
    }
  ```
  Verifica l'esistenza dell'elemento rappresentato dalla chiave contenuta nella variabile '**`$miachiave`**', all'interno dell'hash '**`%miohash`**'. In caso affermativo esegue alcune istruzioni.

24.17.9 Controllo dell'esecuzione del programma

Nel seguito sono elencate le funzioni che sono utili per controllare l'esecuzione di un programma Perl. In particolare ciò che permette di gestire le situazioni di errore.

Tabella 24.210. Funzioni Perl per la gestione delle situazioni di errore.

Funzione	Descrizione
`die` ***lista***	***die()*** emette il contenuto degli elementi della lista fornita come argomento attraverso lo standard error e quindi termina l'esecuzione del programma. Il programma Perl terminato in questo modo restituisce generalmente il valore contenuto dalla variabile '**`$!`**'.
`do` ***file***	***do()*** permette di includere il file indicato come argomento. In generale viene usato per inserire delle subroutine esterne.
`eval` ***blocco*** `eval` ***espressione***	***eval()*** permette di controllare l'esecuzione di un blocco di istruzioni, in modo da limitare i danni in caso di interruzione. In pratica, se all'interno del blocco si manifesta un errore di sintassi o di esecuzione, o ancora se viene incontrata un'istruzione ***die()***, ***eval()*** restituisce un valore indefinito e l'esecuzione del programma continua. Se si manifesta un errore, questo viene riportato dalla variabile '**`$@`**'. Nel caso non si verifichino errori, ***eval()*** restituisce il valore dell'ultima espressione del blocco di istruzioni controllato.
`exit` ***espressione***	***exit()*** valuta l'espressione posta come argomento e termina l'esecuzione del programma restituendo all'esterno quel valore. È importante ricordare che dal punto di vista dei programmi, la restituzione del valore zero corrisponde a una conclusione con successo, mentre un valore pari a uno o superiore, rappresenta una conclusione anomala.
`require` ***espressione*** `require` ***file***	***require()*** permette di specificare nel programma l'esigenza di qualcosa. Se si tratta di un'espressione il cui risultato è numerico, si vuole indicare che il programma richiede un interprete '**`perl`**' di versione maggiore o uguale a quel numero. Se si tratta di una stringa si intende che il programma richiede l'inclusione del file corrispondente come libreria. L'inclusione del file si ottiene solo se ciò non è già avvenuto.
`warn` ***lista***	***warn()*** emette il contenuto degli elementi della lista fornita come argomento attraverso lo standard error. Solitamente, ***warn()*** viene utilizzato come ***die()*** nelle situazioni in cui non è necessario interrompere l'esecuzione del programma.

Segue la descrizione di alcuni esempi.

- ```
  if (chdir '/var/spool/lpd')
    {
      …
    }
  else
    {
      die "L'operazione non è consentita.\n";
    }
  ```
  Se lo spostamento nella directory '`/var/spool/lpd/`' fallisce, visualizza il messaggio attraverso lo standard error e termina.

- ```
  do 'prova.pl';
  ```
  Esegue il contenuto del file '`prova.pl`'.

- ```
  if (chdir '/var/spool/lpd')
    {
      …
    }
  else
    {
      print "L'operazione non è consentita.\n";
      exit 1;
    }
  ```
  Se lo spostamento nella directory '`/var/spool/lpd/`' fallisce, visualizza il messaggio e termina restituendo il valore uno.

24.18 Riferimenti

«

- Johan Vromans, *Perl 5 Desktop Guide*, O'Reilly & Associates, *ftp://ftp.perl.org/pub/CPAN/authors/Johan_Vromans/*

[1] Anche se la documentazione fa esplicito riferimento a questa possibilità, può darsi che non sia possibile evitare di indicare i permessi. Nello stesso modo, anche se si indicano i permessi non è garantito che questi vengano rispettati fedelmente dal sistema operativo sottostante, come descritto nell'esempio successivo.
  ```