



Perl Tutorial

a cura di **Andrea Sivieri**

Versione PDF: Marco Adami (marco.adami@ieee.org)

Questo documento è l'unione degli articoli scritti da Andrea Sivieri, ed intitolati, "Perl Tutorial" pubblicati nei numeri 7,8 e 9 del Journal dello Student Branch di Padova.

Indice

1 Prefazione	3
2 Un programma elementare	4
2.1 Prima linea	4
2.2 Commenti e istruzioni	4
2.3 Semplice stampa a schermo	4
3 Eseguire un programma	5
4 Variabili scalari	6
4.1 Operazioni e assegnazioni	6
4.2 Interpretazione delle stringhe	7
4.3 Esercizio	7
5 Array	8
5.1 Assegnazione di array	8
5.2 Visualizzare array	9
5.3 Esercizio	9
6 Gestione file	10
6.1 Esercizio	11
7 Strutture di controllo	12
7.1 Foreach	12
7.2 Valori di verità	12
7.3 For	13
7.4 While e until	13
7.5 Esercizio	14
8 Condizioni	15
8.1 Esercizio	15
9 Funzioni	16
9.1 Parametri	16
9.2 Ritornare valori	17
9.3 Variabili locali	17
10 Array associativi	18
10.1 Operatori	18
10.2 Variabili d'ambiente	19
11 Ricerca stringhe	20
11.1 Espressioni regolari	20
11.2 La variabile speciale \$_	20
11.3 Di più sulle espressioni regolari	21
11.4 Qualche esempio di ER	22
11.5 Esercizio	23
12 Sostituzioni	24
12.1 Opzioni	24
12.2 Ricordarsi le corrispondenze	25
12.3 Trasformazione di caratteri	26
12.4 Esercizio	27
13 Divisione stringhe	28
13.1 Esercizio	29

Capitolo 1

Prefazione

Il Perl permette di operare su file di testo di qualsiasi tipo in modo molto semplice ed efficace grazie ad una evoluta gestione di stringhe ed espressioni regolari.

Rappresenta quindi una scelta ottima quando si vuole scrivere velocemente un programma che crei o comunque processi pagine html, sorgenti LATEX o file di configurazione. Viene anche spesso utilizzato quando si vogliono estrarre informazioni utili da grossi file di testo, raccolte di messaggi di posta elettronica o grossi file di log generati da qualche programma.

Il Perl è stato ideato da Larry Wall. Egli afferma che il suo linguaggio intende essere pratico (facile da usare, efficiente, completo) piuttosto che bello (leggero, elegante, minimale). Con questa idea in mente egli ha inserito nel Perl molte caratteristiche presenti in altri strumenti software come ad esempio il C, sed, awk e sh. In questo modo ci sono nel Perl molte comodità a portata di mano e si possono superare diversi dei limiti degli altri programmi.

In definitiva il Perl si propone per risolvere problemi di tutti i giorni per i quali non vale la pena di scrivere un programma in C/C++ o in un altro linguaggio compilato tradizionale. Larry Wall scherzosamente dice che le tre principali virtù di un programmatore sono Pigrizia, Impazienza e Hybris.

A partire da questo numero dello IEEE SB PD Journal proporrò ai nostri lettori una traduzione libera non ufficiale di una semplice guida progressiva al Perl scritta da Nik Silver. Spero in questo modo di avvicinare al simpatico linguaggio di programmazione del dromedario le persone interessate. Questi sono i titoli dei dodici capitoletti che costituiscono la guida:

1. *Un programma elementare*
2. *Eseguire un programma*
3. *Variabili scalari*
4. *Array*
5. *Gestione file*
6. *Cicli*
7. *Condizioni*
8. *Array associativi*
9. *Funzioni*
10. *Ricerca stringhe*
11. *Sostituzioni*
12. *Divisione stringhe*

I primi sono presenti nelle pagine successive, gli altri usciranno con i prossimi numeri. Notare che le caratteristiche per cui vale veramente la pena di usare il Perl verranno presentate negli ultimi capitoli, mentre nella prima metà della guida vengono trattate le basi del linguaggio.

La versione originale del tutorial di Nik Silver su cui mi sono basato si trova al seguente indirizzo: <http://www.scs.leeds.ac.uk/Perl/start.html>

Buon divertimento!

Capitolo 2

Un programma elementare

Ecco qui il semplice programma Perl che useremo per iniziare.

```
#!/usr/local/bin/perl
#
# Programma con risultato ovvio
#
print 'Hello world.';
```

Ogni parte verrà discussa separatamente.

2.1 Prima linea

Ogni programma scritto in perl inizia con una riga di questo tipo:

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

anche se ci possono essere delle lievi varianti dipendenti dal sistema. Questa riga indica al sistema operativo cosa fare quando il file che abbiamo scritto viene eseguito (in questo caso vogliamo che il file venga passato al Perl).

2.2 Commenti e istruzioni

I commenti possono venire inseriti in un programma con il simbolo `#`, e tutto a partire dal `#` fino alla fine della riga viene ignorato (ad eccezione della prima riga). L'unico modo per fare commenti multiriga è usare un `#` su ogni riga.

Tolte le righe di commento e la prima riga, rimangono le istruzioni Perl effettive. Ogni istruzione deve essere terminata con un punto e virgola, come mostrato nell'esempio sopra.

2.3 Semplice stampa a schermo

La funzione `print` permette di visualizzare a schermo informazioni testuali. Nel caso che stiamo esaminando viene usata per scrivere la stringa *Hello world*.

Adesso è venuto il momento di eseguire il programma.

Capitolo 3

Eeguire un programma

Dopo aver scritto il programma d'esempio con un editor di testo, salviamolo su disco. Emacs e nedit sono ottimi editor per fare questo, perché tra l'altro hanno una modalità apposta per scrivere in Perl. Come al solito usate l'editor con cui vi trovate meglio. Adesso dobbiamo assicurarci che il programma sia eseguibile utilizzando il comando:

```
chmod u+x NomeProgramma
```

al prompt di UNIX o Linux. Ovviamente al posto di *NomeProgramma* va messo il nome effettivo che abbiamo usato per il file contenente il programma. Adesso per lanciare il programma non ci rimane che digitare una delle seguenti cose al prompt:

```
perl NomeProgramma  
./NomeProgramma  
NomeProgramma
```

Se qualcosa va storto, allora potreste ottenere come risultato messaggi di errore oppure niente. Per eseguire il programma attivando tutti i messaggi di errore e di avvertimento basta usare il comando perl in questo modo:

```
perl -w NomeProgramma
```

Questo mostrerà avvertimenti e altri utili (si spera) messaggi già prima che parta l'esecuzione vera e propria del programma. Invece per eseguire il programma con un debugger usare:

```
perl -d NomeProgramma
```

Quando si manda in esecuzione un file Perl, prima esso viene compilato e poi viene eseguita la versione compilata. Per questa ragione dopo una piccola pausa iniziale, il programma dovrebbe girare abbastanza veloce. Questo spiega come possa essere "eseguito" un file che contiene esclusivamente testo. Assicuratevi che il programma giri correttamente prima di procedere.

Capitolo 4

Variabili scalari

Una variabile scalare è il tipo più elementare di variabile in Perl. Le variabili scalari possono contenere sia stringhe che numeri, inoltre stringhe e numeri sono completamente interscambiabili. Per esempio, l'espressione

```
$urgenza = 9;
```

imposta la variabile scalare *\$urgenza* a 9, ma potete anche tranquillamente assegnare una stringa alla stessa identica variabile:

```
$urgenza = 'grande';
```

Perl accetta i numeri anche come stringhe, come in questo caso:

```
$urgenza = '9';
$contatore = '0009';
```

e possono ancora essere eseguite operazioni aritmetiche senza particolari problemi. In generale i nomi di variabile sono costituiti da numeri, lettere e da alcune eventuali occorrenze del carattere underscore . Bisogna però tenere conto che i nomi scelti non devono iniziare con un numero e che la variabile `$_` è speciale, come vedremo più avanti. Infine è da osservare che Perl distingue tra minuscole e maiuscole, quindi `$a` e `$A` sono differenti.

4.1 Operazioni e assegnazioni

Perl utilizza tutti gli usuali operatori aritmetici del C:

```
$a = 1 + 2;      # Somma 1 e 2 e salva il risultato in $a
$a = 3 - 4;     # Sottrae 4 da 3 e salva in $a
$a = 5 * 6;     # Moltiplica 5 e 6
$a = 7 / 8;     # Divide 7 per 8 ottenendo 0.875
$a = 9 ** 10;  # Nove alla decima
$a = 5 % 2;     # Il resto di 5 diviso 2
++$a;          # Incrementa di uno $a, poi restituisce il suo valore
$a++;          # Restituisce il valore di $a, poi incrementa $a
--$a;          # Decrementa, poi restituisce
$a--;          # Restituisce, poi decrementa
```

Per le stringhe sono inoltre disponibili i seguenti operatori:

```
$a = $b . $c;   # Concatena $b e $c
$a = $b x $c;   # $b ripetuta $c volte
```

Per assegnare un valore:

```
$a = $b;           # Assegna il valore $b ad $a
$a += $b;         # Incrementa $a di $b unità
$a -= $b;         # Sottrae $b da $a
$a .= $b;         # Appende $b alla fine di $a
```

Osservare che per fare un'assegnazione del tipo `$a=$b` viene fatta una copia di `$b` e poi assegnata tale copia ad `$a`. Per questa ragione la prossima volta che verrà modificata `$b` non verrà alterata `$a`.

Altre operazioni ancora possono essere trovate nella pagina di manuale perlop:

`man perlop`

4.2 Interpretazione delle stringhe

Il seguente pezzo di codice scrive mele e pere usando la concatenazione:

```
$a = 'mele';
$b = 'pere';
print $a.' e '.$b;
```

Sarebbe bello passare una unica stringa quando si chiama la funzione di scrittura, ma la riga

```
print '$a e $b';
```

scrive in modo letterale `$a` e `$b`, invece di mele e pere. Per ottenere questo ultimo risultato scrivere:

```
print "$a and $b";
```

Le doppie virgolette forzano l'interpretazione di tutti i codici presenti all'interno, i nomi di variabile ad esempio vengono sostituiti con il contenuto associato. Altri codici che vengono sostituiti sono i caratteri speciali come `newline` e `tab`. Il codice `\n` rappresenta `newline` e `\t` invece `tab`.

4.3 Esercizio

Riscrivere il programma *Hello world* in modo che:

1. la stringa sia assegnata ad una variabile
2. sia stampato il contenuto della variabile seguito da un carattere di `newline`.

Usare le doppie virgolette invece che l'operatore di concatenazione.

Capitolo 5

Array

Un tipo interessante di variabile è l'array, che non è altro che una sequenza di scalari (numeri o stringhe). Le variabili array hanno lo stesso formato delle variabili scalari eccetto per il fatto che sono precedute dal simbolo `@` invece che `$`. L'espressione:

```
@cibo = ("mele", "pere", "uva");
@musica = ("viola", "flauto");
```

assegna una sequenza di tre elementi alla variabile array `@cibo` e una sequenza di due elementi a `@musica`.

Si accede all'array utilizzando indici che partono da 0. L'indice viene racchiuso tra parentesi quadre. L'espressione

```
$cibo[2]
```

ritorna *uva*. Notare che al posto di `@` si è messo `$`, perché *uva* è uno scalare.

5.1 Assegnazione di array

Come sempre in Perl, la stessa espressione in contesti diversi può produrre risultati differenti. La prima assegnazione qui sotto considera i valori scalari estratti da `@musica` ottenendo in questo caso lo stesso risultato della seconda assegnazione.

```
@altramusica = ("organo", @musica, "arpa");
@altramusica = ("organo", "viola", "flauto", "arpa");
```

Questo suggerisce un modo per aggiungere elementi ad un array. Un modo più pulito di aggiungere elementi è usare l'istruzione

```
push(@cibo, "uovo");
```

che aggiunge *uovo* alla fine dell'array `@cibo`. Per aggiungere due o più elementi all'array ci sono ad esempio le seguenti alternative:

```
push(@cibo, ("uovo", "pane"));
push(@cibo, @altrocibo);
```

La funzione `push` ritorna la lunghezza dell'array dopo l'espansione. Per rimuovere l'ultimo elemento da un array e ritornarlo usare la funzione `pop`. Ecco `pop` prelevare *pane*, poi `$cibo` ha due elementi:

```
$fame = pop(@cibo);    # Ora $fame = "pane"
```

È anche possibile assegnare un array ad una variabile scalare. Il contesto è importante. La riga

```
$f = @cibo;
```

assegna a *\$f* la lunghezza di *@cibo*, ma

```
$f = "@cibo";
```

sostituisce al nome dell'array una stringa che rappresenta i suoi elementi separati fra di loro da spazi. Al posto dello spazio di separazione si può mettere qualsiasi altra stringa cambiando il valore della variabile speciale *\$*. Questa variabile è una delle tante variabili speciali del Perl, la maggior parte delle quali ha nomi strani.

Gli array possono anche essere usati per fare assegnazioni multiple a variabili scalari:

```
($a, $b) = ($c, $d);      # Lo stesso che fare $a=$c; $b=$d;
($a, $b) = @cibo;        # $a e $b sono i primi due elementi di @cibo
($a, @delcibo) = @cibo;  # $a è il primo elemento di @cibo
                        # @delcibo è un array contenente gli altri
(@delcibo, $a) = @cibo;  # @delcibo è uguale a @cibo, mentre $a non è definita
```

Osservare che gli array sono ingordi e che nell'ultima assegnazione *\$delcibo* prenderà così tanto *@cibo* quanto può, lasciando *\$a* non definita. È meglio quindi evitare questa forma.

Infine potreste voler trovare l'indice dell'ultimo elemento di un array. Nel caso dell'array *@cibo* si può trovare tale valore nella seguente variabile associata ad esso:

```
 $#cibo
```

5.2 Visualizzare array

Dal momento che il contesto è importante, non dovrebbe essere sorprendente che le seguenti righe producono ognuna un risultato diverso:

```
print @cibo;           # da solo
print "@cibo";        # circondato da doppie virgolette
print $cibo."";       # in un contesto scalare
```

5.3 Esercizio

Provare ognuna delle precedenti modalità di stampa per vedere quali sono le differenze di comportamento.

Capitolo 6

Gestione file

Qui di seguito un programma perl elementare che fa le stesse cose del comando UNIX `cat` su di un particolare file. Come già osservato in precedenza perché tutto funzioni bene la prima riga deve corrispondere alla configurazione del sistema su cui si sta operando; su molti sistemi ad esempio l'interprete perl non si trova in `/usr/local/bin/`, ma invece in `/usr/bin`.

```
#!/usr/local/bin/perl
#
# Programma per aprire il file pippo.txt, leggerlo,
# stamparlo a schermo e infine chiuderlo.
# Nome del file da leggere
#
$file = 'pippo.txt';
open(INFO, $file);           # Apre il file
@righe = <INFO>              # Legge tutto in un array
close(INFO);                # Chiude il file
print @righe;               # Visualizza l'array
```

La funzione `open` apre un file in lettura. Il primo argomento è l'identificatore di file che permetterà in seguito di riferirsi al file aperto. Il secondo argomento è invece il nome del file da aprire. Se il nome viene dato racchiuso tra virgolette singole, allora esso viene usato in modo letterale, senza che venga fatta l'espansione da parte della shell. Ad esempio l'espressione `'~/note/CoseDaFare.txt'` non verrebbe interpretata correttamente. Se si desidera l'interpretazione da parte della shell usare invece le parentesi angolate in questo modo: `<~/note/CoseDaFare.txt>`.

La funzione `close` dice al Perl che non ci serve più l'accesso a quel file. Ci sono alcune cose utili da aggiungere a proposito della gestione dei file. Prima di tutto la funzione `open` può anche specificare un file da usare per salvare o appendere i dati in uscita, oltre che per leggere dati in ingresso:

```
open(INFO, $file);          # Apre in lettura
open(INFO, ">$file");        # Apre in scrittura
open(INFO, ">>$file");       # Apre per appendere l'uscita
open(INFO, "<$file");         # Altro modo per aprire in lettura
```

In secondo luogo, se si vuole scrivere qualcosa su di un file già aperto in scrittura, allora si può usare il comando `print` con un argomento addizionale. Ad esempio per scrivere una stringa sul file con l'identificatore `INFO` usare

```
print INFO "Scrivo questa riga nel file.\n";
```

A questo punto osserviamo che in modo simile si possono aprire anche lo standard

input (generalmente associato con la tastiera) e lo standard output (generalmente lo schermo):

```
open(INFO_IN, '-');      # Apre lo standard input
open(INFO_OUT, '>-');    # Apre lo standard output
```

Nel programma visto sopra le informazioni sono lette da un file. L'identificatore di tale file è INFO e si legge da esso usando le parentesi angolate. Così l'istruzione

```
@lines = <INFO>
```

legge il file associato a INFO nell'array &righe. Notare che con questa tecnica il file viene letto tutto in una volta. Questo succede perché la lettura avviene nel contesto di una variabile array. Se invece &righe venisse sostituito dalla variabile scalare \$riga, allora solo una riga verrebbe letta. In entrambi i casi ogni riga viene memorizzata completa con il suo carattere di newline alla fine.

6.1 Esercizio

Modificare il programma appena esaminato in modo che l'intero file venga stampato con il simbolo # all'inizio di ogni riga. È possibile farlo aggiungendo una riga e modificandone un'altra. Provare a usare la variabile speciale \$". Nel caso si presentassero problemi inaspettati, potrebbe essere utile lanciare il Perl con l'opzione -w, come già visto nella sezione su come eseguire un programma.

Capitolo 7

Strutture di controllo

Le possibilità più interessanti nascono quando introduciamo strutture di controllo e cicli. Il Perl supporta molti tipi di queste strutture e si possono notare grosse somiglianze con i costrutti simili in C e Pascal. Vediamo ora i casi più importanti.

7.1 Foreach

Per prendere in considerazione una alla volta ogni riga di un array o di un altro contenitore di tipo lista (come ad esempio le diverse righe in un file di testo) in Perl si usa il comando `foreach`. Così ad esempio:

```
foreach $boccone (@cibo)      # Visita un elemento per volta e chiamalo $boccone
{
print "$boccone\n";          # stampa l'elemento
print "Yum yum\n";          # si possono fare anche altre cose...
}
```

Le azioni che devono essere eseguite ogni volta sono racchiuse in un blocco individuato dalle parentesi graffe. La prima volta che il blocco viene eseguito `$boccone` contiene il primo elemento dell'array `$cibo`. La volta successiva a `$boccone` viene assegnato l'elemento successivo dell'array e così via. Se quando si parte `&cibo` è vuoto, allora il blocco di istruzioni interne non viene mai eseguito.

7.2 Valori di verità

Le seguenti strutture dipenderanno dai risultati di una valutazione logica. In Perl ogni numero diverso da zero e ogni stringa non vuota sono considerati avere il valore logico di vero. Al contrario il numero zero, il carattere zero stesso presente isolato in una stringa e una stringa vuota sono considerati avere il valore di falso.

```
$a == $b      # E' $a numericamente uguale a $b?
              # (Attenzione: non usare l'operatore =)
$a != $b      # E' $a numericamente diverso da $b?
$a eq $b      # E' $a uguale come stringa a $b?
$a ne $b      # E' $a diverso come stringa da $b?
```

Si possono inoltre usare anche gli operatori logici `and`, `or` e `not`:

```
($a && $b)    # Sono $a e $b contemporaneamente veri?
($a || $b)    # E' vero almeno uno dei due?
!($a)         # $a è falso?
```

7.3 For

Il Perl ha una struttura for molto simile a quella del C. Nelle parentesi che seguono il for vanno messe tre espressioni separate da punti e virgola. Come prima espressione si mette il comando di inizializzazione; esso verrà eseguito solo all'inizio, prima di entrare nel ciclo vero e proprio. La seconda espressione rappresenta invece la condizione di continuazione; prima dell'esecuzione di ogni ciclo essa viene valutata e se risulta falsa si esce dal ciclo, altrimenti si continua. Infine la terza espressione viene eseguita alla fine di ogni iterazione del ciclo, dopo che sono state eseguite tutte le istruzioni specificate nel blocco interno.

Ecco un esempio di ciclo che stampa i numeri da 0 a 9:

```
for ($i = 0; $i < 10; ++$i)      # Inizia con $i = 1, fallo se $i < 10, ...
{                                # .... incrementa $i, prima di ripetere
    print "$i\n";
}
```

7.4 While e until

Qui di seguito viene presentato un programma che legge un ingresso dalla tastiera e non prosegue finché non viene inserita la parola corretta.

```
#!/usr/local/bin/perl
print "Password? ";           # Richiedi la password
$a = <STDIN>                  # Metti l'ingresso in $a
chop $a;                      # Rimuovi il newline finale
while ($a ne "pippo")        # Finché l'ingresso è sbagliato...
{                              # ... ripeti la richiesta
    print "Sbagliato: riprova! "; # Leggi di nuovo l'ingresso
    $a = <STDIN>
    chop $a;
}
```

Il blocco di codice individuato dalle parentesi graffe è eseguito finché l'ingresso non è uguale a "pippo". Il funzionamento di while dovrebbe risultare chiaro, abbiamo però l'opportunità di osservare diverse cose. Primo, possiamo leggere dallo standard input (la tastiera) senza aprire prima il file. Secondo, quando la password viene inserita \$a assume il valore di quella stringa incluso il carattere di newline alla fine. La funzione chop serve per rimuovere l'ultimo carattere di una stringa, in questo caso newline.

Eventualmente al posto di while si può usare until, tenendo presente che until(\$condizione) equivale a while(!(\$condizione)). Nell'ultimo programma visto bisognerebbe mettere until(\$a eq "pippo") al posto di while(\$a ne "pippo"), perché il programma si comporti in modo identico.

Un'altra tecnica utile è mettere il controllo while o until alla fine del blocco di istruzioni del ciclo invece che all'inizio. Questo richiederà la presenza dell'operatore do all'inizio per segnalare che il controllo della condizione sarà fatto alla fine.

Se escludiamo il messaggio "Sbagliato: riprova! " allora il programma precedente

potrebbe essere riscritto in questo modo:

```
#!/usr/local/bin/perl
do
{
    print "Password? ";           # Chiedi l'ingresso
    $a = <STDIN>                  # Leggi l'ingresso
    chop $a;                      # Elimina il newline
}
while ($a ne "pippo")           # Se l'ingresso è sbagliato, ripeti
```

7.5 Esercizio

Modificare il programma relativo all'esercizio del capitolo precedente in modo che ogni linea del file sia letta una alla volta e sia data in uscita preceduta da un numero di riga.

Si dovrebbe ottenere qualcosa del genere:

```
1 root:oYpYXm/qRO6N2:0:0:Super-User:./bin/csh
2 sysadm:*:0:0:System V Administration:/usr/admin:/bin/sh
3 diag:*:0:996:Hardware Diagnostics:/usr/diags:/bin/csh
etc
```

Potrà essere utile il seguente costrutto:

```
while ($line = <INFO>)
{
...
}
```

Ottenuto il risultato cercato provare a modificare il programma in modo che i numeri di riga siano scritti come 001, 002, ..., 009, 010, 011, 012, etc. Per fare questo dovrebbe essere sufficiente alterare una riga inserendo quattro caratteri aggiuntivi. Con il perl è generalmente facile trovare scorciatoie di questo tipo...

Capitolo 8

Condizioni

Ovviamente il Perl consente anche le espressioni `if / then / else`. Ecco il modo in cui utilizzarle:

```
if ($a)
{ print "La stringa non è vuota\n";
}
else
{ print "La stringa è vuota\n";
}
```

Per capire il codice sopra ricordiamo che una stringa vuota è giudicata falsa. Allo stesso modo la stringa `0` è falsa e quindi nel codice appena visto darebbe lo stesso risultato di una stringa vuota.

Oltretutto è possibile fornire diverse alternative in un costrutto condizionale:

```
if (!$a)                                # il ! è l'operatore negazione
{ print "La stringa è vuota\n";
}
elsif (length($a) == 1)                 # altrimenti se ...
{ print "La stringa contiene un carattere\n";
}
elsif (length($a) == 2)                 # altrimenti se ...
{ print "La stringa contiene due caratteri\n";
}
else                                     # altrimenti
{ print "La stringa ha molti caratteri\n";
}
```

Importante notare che sopra si è usato `elsif`, non `elseif`. Insomma non si tratta di un errore di battitura.

8.1 Esercizio

Trovare un file piuttosto grande che contenga del testo e delle righe vuote.

Dall'ultimo esercizio svolto si dovrebbe avere un programma che stampa il file delle password con i numeri di riga. Cambiare il programma in modo che lavori sul nuovo file di testo scelto. Fare in modo che le righe vuote vengano visualizzate, ma che esse non siano precedute dal numero di riga e non vengano contate come le altre. Ricordare che quando la riga di un file viene letta essa contiene ancora alla fine il carattere di newline.

Capitolo 9

Funzioni

Come ogni buon linguaggio di programmazione anche il Perl permette di definire funzioni. Esse possono essere collocate ovunque nel programma, ma è probabilmente meglio metterle tutte all'inizio o tutte alla fine. Una funzione ha la forma

```
sub funzione_mia
{
    print "Funzione non molto interessante:\n";
    print "Ogni volta fa la stessa cosa!\n";
}
```

indipendentemente dagli eventuali parametri che si potrebbero passare. Ognuna delle seguenti scritture risulta valida per chiamare una funzione; notare che va messa una & prima del nome effettivo quando si effettua una chiamata.

```
&funzione_mia;           # Chiamata semplice
&funzione_mia($_);      # Chiamata con un parametro
&funzione_mia(1+2, $_); # Chiamata con due parametri
```

9.1 Parametri

Nei casi precedenti i parametri erano accettabili, ma ignorati. Quando la funzione è chiamata tutti i parametri sono passati come una lista array nella variabile speciale @_. Questa variabile non ha nessuna relazione con la variabile scalare \$_. La seguente funzione visualizza banalmente la lista dei parametri con i quali è stata invocata. Seguono un paio di esempi di chiamate.

```
sub scrivi_param
{
    print "@_\n";
}
&scrivi_param("re", "nudo");
&scrivi_param("rana", "e", "bue");
```

Esattamente come con qualsiasi altra lista array, si può accedere ai singoli elementi di @_ con l'operatore parentesi quadre.

```
sub scrivi_i_primi_due
{
    print "Il primo argomento è $_[0]\n";
    print "e $_[1] è il secondo\n";
}
```

Di nuovo bisogna sottolineare che gli scalari `$_[0]` e `$_[1]` e così via non hanno niente a che fare con lo scalare `$_` che può sempre venire usato senza timore di conflitti.

9.2 Ritornare valori

Il valore di ritorno di una funzione è sempre l'ultima cosa valutata. Questa funzione ritorna il massimo dei due parametri in ingresso. Segue un esempio del suo uso.

```
sub massimo
{
  if ($_[0] > $_[1])
  {
    $_[0];
  }
  else
  {
    $_[1];
  }
}
$max = &massimo(37, 24);
```

La funzione `&scrivi_i_primi_due` a sua volta ritorna un valore, precisamente 1. Questo dipende dal fatto che l'ultima istruzione della funzione è il comando `print` e il risultato di un comando `print` che ha avuto successo è sempre 1.

9.3 Variabili locali

La variabile `@_` è locale alla funzione corrente, lo stesso vale ovviamente per `$_[0]`, `$_[1]`, `$_[2]` e così via. Si possono rendere locali anche altre variabili, e questo è utile se vogliamo alterare i parametri in ingresso. La seguente funzione testa per vedere se una stringa è dentro ad un'altra, senza contare gli spazi. Segue un esempio di chiamata.

```
sub dentro
{ local($a, $b);           # rendi le due variabili locali
  ($a, $b) = ($_[0], $_[1]); # assegna i valori
  $a =~ s/ //g;           # elimina gli spazi
  $b =~ s/ //g;
  ($a =~ /$b/ || $b =~ /$a/); # si ha che $a contiene $b oppure che $b contiene $a?
}
&dentro("casa", "marca sapone");
# viene ritornato vero
```

A dire il vero le prime righe potevano essere raggruppate in una riga sola, in questo modo:

```
local($a, $b) = ($_[0], $_[1]);
```

Capitolo 10

Array associativi

Le normali liste array permettono di accedere ai loro elementi specificando un numero. Infatti il primo elemento dell'array `@cibo` è `$cibo[0]`, il secondo è `$cibo[1]` e così via. Il Perl daltronde permette anche di creare array i cui elementi sono accessibili specificando delle stringhe. Strutture dati di questo tipo sono dette array associativi. Per definire un array associativo si usa la solita notazione con le parentesi, ma in questo caso il nome dell'array viene preceduto dal simbolo `%`. Supponiamo di voler creare un array contenente i nomi di diverse persone e le loro età. Un modo possibile è il seguente:

```
%info_persone = ("Michele Perlato", 39, "Marco Rossi", 34, "Rodolfo", "3 anni da pesce",
                "Guglielmo De Filippis", 21, "Artemide Dal Canto", 108);
```

Ora possiamo trovare l'età delle persone (e del pesce) in questo modo:

```
$info_persone{"Michele Perlato"};      # ritorna 39
$info_persone{"Marco Rossi"};        # ritorna 34
$info_persone{"Guglielmo De Filippis"}; # ritorna 21
$info_persone{"Artemide Dal Canto"};  # ritorna 108
$info_persone{"Rodolfo"};            # ritorna "3 anni da pesce"
```

Notare che come nel caso delle liste array il segno `%` è stato cambiato nel segno `$` per accedere ad un singolo elemento, perché un singolo elemento è uno scalare. Al contrario delle liste array l'indice identificativo (in questo caso il nome della persona) è racchiuso tra parentesi graffe, dal momento che gli array associativi sono più fantasiosi rispetto a quelli classici.

Se si assegna l'array associativo `%info_persone` alla lista array `&info`, si ottiene una lista array contenente 10 elementi. Gli elementi di indice pari sono costituiti dai nomi, quelli di indice dispari dalle età; l'abbinamento originario tra nomi ed età viene mantenuto. Viceversa è possibile in modo semplice convertire una lista array in un array associativo. I seguenti esempi chiariranno quanto detto:

```
@info = %info_persone;      # @info è una lista array contenente 10 elementi
$info[2];                   # ritorna "Marco Rossi"
$info[3];                   # ritorna 34
$info_persone2 = @info;    # %info_persone2 è ora un array
                           # associativo uguale a %info_persone
```

10.1 Operatori

Non è possibile attribuire un certo ordinamento agli elementi di un array associativo, come è invece possibile fare con gli elementi di un array classico. Risulta comunque semplice accedere a tutti gli elementi uno per volta usando le funzioni `key` e `values`:

```
foreach $persona (keys %info_personone)
{
    print "Conosco l'età di $persona\n";
}
foreach $anni (values %info_personone)
{
    print "Qualcuno ha $anni anni\n";
}
```

La funzione `keys` ritorna una lista di tutte le chiavi (indici simbolici) dell'array associativo. Invece `values` restituisce la lista di tutti i valori memorizzati nell'array associativo specificato. Queste due funzioni ritornano le loro liste nello stesso ordine, ma questo ordine non ha niente a che vedere con lo'ordine in cui gli elementi sono stati inseriti. Quando `keys` e `values` sono chiamate in un contesto scalare esse ritornano il numero di coppie chiave/valore contenute nell'array associativo. Un'altra funzione interessante è `each`. Essa restituisce ogni volta che viene chiamata una coppia diversa di elementi; il primo elemento è una chiave, il secondo è il valore relativo.

```
while (($persona, $anni) = each(%info_personone))
{
    print "$persona ha $anni anni\n";
}
```

10.2 Variabili d'ambiente

Quando si esegue un programma Perl o qualsiasi altro script UNIX, sarà spesso utile conoscere il valore a cui sono state impostate tutta una serie di variabili d'ambiente. Ad esempio la variabile `USER` contiene il nome di login dell'utente che sta eseguendo lo script, la variabile `DISPLAY` indica su quale schermo andrà a finire l'output grafico del programma. Quando poi si esegue uno script CGI perl attraverso Internet, ci sono svariate altre variabili d'ambiente che contengono informazioni utili. Tutte queste variabili e i loro rispettivi valori sono memorizzati nell'array associativo `%ENV`; il nome delle variabili è ovviamente la chiave per accedere al valore associato. Ecco un semplice esempio di utilizzo:

```
print "Ti chiami $ENV{'USER'} e stai usando ";
print "il display $ENV{'DISPLAY'}\n";
```

Capitolo 11

Ricerca stringhe

11.1 Espressioni regolari

Un'espressione regolare viene racchiusa tra due barre oblique (slash), e per vedere se esiste una corrispondenza tra una stringa ed una certa ER si usa l'operatore `=~`. La seguente espressione è vera se la sequenza di caratteri *la* compare nella variabile *\$frase*.

```
$frase =~ /la/
```

L'ER è sensibile alle minuscole e alle maiuscole, così se

```
$frase = "La veloce volpe rossa";
```

allora il confronto provato in precedenza darà esito negativo. Invece l'operatore `!~` è usato per rilevare una non corrispondenza. Rispetto all'esempio sopra

```
$frase !~ /la/
```

è vera, perché la sequenza *la* non compare in *\$frase*.

11.2 La variabile speciale `$_`

Potremmo usare espressioni condizionali tipo

```
if( $frase =~ /goal/ )
{ print "Stiamo parlando di calcio\n";
}
```

che scriverebbe un messaggio per esempio in uno dei seguenti casi:

```
$frase = "Siamo sotto di un goal.";
$frase = "Un altro goal: evviva!".
```

Ma è spesso molto più semplice se assegnamo la stringa alla variabile speciale `$_`, che è ovviamente uno scalare. Se facciamo ciò possiamo evitare di usare gli operatori `=~` e `!~` e scrivere semplicemente:

```
if( /goal/ )
{ print "Stiamo parlando di calcio\n";
}
```

La variabile `$_` è la variabile predefinita usata da molti operatori Perl e tende ad essere sfruttata molto spesso per questo fatto.

11.3 Di più sulle espressioni regolari

In una ER ci sono parecchi caratteri speciali, e sono questi che allo stesso tempo danno loro tanto potere e le fanno sembrare complicate. È meglio iniziare ad usarle un po' alla volta; la loro creazione può essere a volte una forma d'arte.

Ecco qui alcuni caratteri speciali ed il loro significato

.	# Qualsiasi carattere singolo escluso newline
^	# Inizio di stringa o di riga
\$	# Fine di stringa o di riga
*	# Zero o più occorrenze dell'ultimo carattere
?	# Zero o una occorrenza dell'ultimo carattere

e adesso invece esempi di corrispondenze possibili. Si ricordi che le ER vanno inserite tra due barre oblique per essere usate.

o.a	# "o" seguita da qualsiasi carattere seguito da "a". # Soddisfatta da ora, oca, osa, ma non da oa, orma.
^f	# "f" all'inizio di una riga
^yogi	# yogi all'inizio di una riga
e\$	# "e" alla fine di una riga
notte\$	# notte a fine riga
aiuto*	# aiuto seguito da zero o più caratteri "o" # Soddisfatta da aiut, aiuto, aiutoo, aiutooo (etc.)
.*	# Qualsiasi stringa senza un newline. Questo perché . si # abbina a qualsiasi carattere fuorché newline e * significa # zero o più di questi.
^\$	# Una riga vuota.

Ci sono anche altre possibilità. Le parentesi quadre sono usate per indicare che si vuole la corrispondenza con uno qualsiasi dei caratteri contenuti. All'interno di queste parentesi per inserire velocemente un'intera sequenza di caratteri consecutivi si può usare il simbolo - posizionato tra i caratteri estremi, inoltre si può usare il simbolo ^ se si desidera esprimere una negazione:

[qjk]	# Alternativamente q oppure j oppure k
[^qjk]	# Né q, né j, né k
[a-z]	# Qualsiasi carattere tra a e z inclusi
[^a-z]	# Nessuna lettera minuscola
[a-zA-Z]	# Qualsiasi lettera
[a-z]+	# Qualsiasi sequenza non vuota di lettere minuscole

Sapendo queste cose si potrebbe già andare alla fine del capitolo ed essere in grado di fare la maggior parte degli esercizi. Comunque per sfruttare al massimo la potenza delle ER, conviene considerare anche la parte che segue.

Una barra verticale | rappresenta un "oppure" e invece una coppia di parentesi () può essere usata per raggruppare assieme parti di ER quando necessario:

```
arancia|limone    # Alternativamente arancia o limone
cas(a|er)ma      # casa oppure caserma
(la)+            # la oppure lala oppure lalala oppure ...
```

Qui altri caratteri speciali utili:

```
\n              # newline
\t              # tab
\w              # Qualsiasi carattere alfanumerico (equivale a [^a-zA-Z0-9_])
\W              # Qualsiasi carattere non alfanumerico (equivale a [^a-zA-Z0-9_])
\d              # Qualsiasi cifra (uguale a [0-9])
\D              # Qualsiasi carattere non cifra (uguale a [^0-9])
\s              # Qualsiasi carattere di spazio: spazio, tab, newline, etc.
\S              # Qualsiasi carattere che non è di tipo spazio
\b              # Confine di parola, si può usare solo fuori da []
\B              # Non confine di parola
```

Chiaramente caratteri tipo \$, |, [,], \, / e così via sono casi particolari nelle espressioni regolari. Se si desidera richiedere una corrispondenza con uno di questi, allora bisogna farli precedere dal simbolo barra rovescia (backslash) in questo modo:

```
\|              # Barra verticale
\|              # Quadra aperta
\|              # Tonda chiusa
\*              # Asterisco
\^              # Casetta (carat)
\/              # Barra (slash)
\\              # Barra rovescia (backslash)
```

e così via.

11.4 Qualche esempio di ER

Infine ecco una lista di esempi di complessità crescente per prendere confidenza con le espressioni regolari. Si ricordi che quando le si usa bisogna farle precedere e seguire da barre (slash) di separazione.

```
[01]           # Alternativamente "0" oppure "1"
\/0            # Una divisione per zero: "/0"
\/0           # Una divisione per zero con spazio: "/ 0"
\/\s0         # Una divisione per zero con spazio generico
\/ *0         # Divisione per zero con quantità arbitraria di spazi normali
\/\s*0        # Divisione per zero con quantità arbitraria di spazi generici
\/\s*0\.0*    # Come prima, ma eventualmente con punto decimale e
              # ulteriori zeri dopo di esso. Vengono accettati "/0." e "/0.0"
              # e "/0.00" etc. e "/ 0." e "/ 0.00" etc.
```

11.5 Esercizio

Scrivere un programma che numera progressivamente tutte e sole le righe in entrata che contengono la lettera "v".

Modificarlo in modo da prendere in considerazione le righe contenenti:

- la stringa "una";
- la stringa "una" con iniziale maiuscola o minuscola;
- la parola "una" con o senza maiuscola iniziale. Usare \ per individuare i confini di parola.

In ogni caso il programma dovrebbe riportare in uscita ogni riga data in entrata, ma numerare solo quelle che rispettano la caratteristica desiderata. Provare a sfruttare la variabile \$_ per evitare di usare esplicitamente l'operatore di confronto =~.

Capitolo 12

Sostituzioni

Oltre a poter trovare se una stringa soddisfa una particolare espressione regolare, il Perl permette di fare sostituzioni basate sulle corrispondenze individuate. Il modo per fare questo è usare la funzione `s`. Ancora una volta viene utilizzato l'operatore `=~`, e ancora una volta se esso viene omissso è presa in considerazione la variabile `$_`. Per sostituire un'occorrenza di `bologna` con `Bologna` nella stringa `$frase` usiamo l'espressione

```
$frase =~ s/bologna/Bologna/
```

e per fare la stessa cosa con la variabile `$_` basta invece

```
s/bologna/Bologna/
```

Si noti che le due espressioni regolari (`bologna` e `Bologna`) sono circondate da un totale di tre barre (slash). Il risultato di questa espressione è il numero di sostituzioni fatte, in questo caso 0 oppure 1.

12.1 Opzioni

L'esempio appena visto sostituisce solamente la prima occorrenza della stringa, ma potrebbe essere che ci siano un numero maggiore di stringhe uguali che desideriamo sostituire. Per fare una sostituzione globale basta mettere una `g` dopo l'ultima barra, così:

```
s/bologna/Bologna/g
```

Ovviamente adesso stiamo operando sulla variabile `$_`. Anche in questo caso l'espressione ritorna il numero di sostituzioni fatte, il valore può essere adesso qualsiasi intero maggiore o uguale a zero.

Se si vuole essere in grado di sostituire anche occorrenze di `bOlogna`, `boLOGNA`, `BoLOgnA` e così via si può usare qualcosa tipo

```
s/[Bb][Oo][Ll][Oo][Gg][Nn][Aa]/Bologna/g
```

ma un modo più semplice è invece usare l'opzione `i` (che sta per "ignora differenza minuscole/minuscole"). L'espressione

```
s/bologna/Bologna/gi
```

effettuerà una sostituzione globale in tutti i casi voluti. L'opzione `i` può essere usata anche quando si fa un normale confronto come quelli visti nel capitolo precedente.

12.2 Ricordarsi le corrispondenze

Spesso è utile ricordarsi quali sottostringhe hanno trovato corrispondenza con certe parti dell'espressione regolare, per poterle riutilizzare. Succede semplicemente che ogni sottostringa che corrisponde ad una parte di espressione regolare racchiusa dentro parentesi tonde viene memorizzata nelle variabili \$1, ..., \$9. Queste stringhe possono anche essere usate nella stessa espressione regolare (o sostituzione) attraverso i codici speciali \1, ..., \9. Per esempio:

```
$_ = "Marchese Alfonso De Filippis";
s/([A-Z]):\1:/g;
print "$_\n";
```

sostituirà ogni lettera maiuscola con la lettera stessa circondata da due punti. Si otterrà insomma ":M:archese :A:lfonso :D:e :F:ilippis". Le variabili \$1, ..., \$9 non sono modificabili: sono a sola lettura.

Come ulteriore esempio, le righe:

```
if (/(\b.+ \b) \1/ )
{
    print "Trovata $1 ripetuta.\n";
}
```

individuano qualsiasi parola ripetuta. Ogni \b rappresenta un confine di parola e .+ corrisponde ad una qualsiasi stringa non vuota, così \ b.+ \b corrisponde ad una qualsiasi parola. Grazie alle parentesi () la parola che soddisfa la corrispondenza viene ricordata nella variabile speciale \1 dentro all'ER oppure nella variabile \$1 per le righe di programma successive.

La seguente espressione scambia il primo e l'ultimo carattere nella variabile \$_:

```
s/^(.)(.*)($)\2\1/
```

I simboli ^ e \$ corrispondono rispettivamente all'inizio ed alla fine della riga. Il codice speciale \1 conserva il primo carattere; \2 conserva tutto il resto fuorché l'ultimo carattere, che viene memorizzato in \3. A questo punto l'intera riga viene sostituita con una versione in cui \1 e \3 sono scambiati.

Dopo aver cercato una corrispondenza, si possono usare le variabili speciali a sola lettura `\$` e `\$\$` e `\$'` per trovare la sottostringa che precede la corrispondenza, quella che soddisfa la corrispondenza e quella che segue. Così dopo

```
$_ = "Marchese Alfonso De Filippis";
/fo/;
```

le seguenti espressioni sono tutte vere. (Si ricordi che eq è l'operatore di eguaglianza che confronta due variabili considerandole stringhe.)

```
$` eq "Marchese Al";
$$ eq "fo";
```

```
$' eq "nso De Filippis";
```

Infine in questa sezione vale la pena di ricordare che all'interno delle barre di separazione delle corrispondenze o sostituzioni le variabili sono sostituite con il loro contenuto. Così

```
$ricerca = "la";  
s/$ricerca/xxx/g;
```

sostituirà ogni occorrenza di "la" con "xxx". Se si desidera invece sostituire ogni occorrenza di "lavagna", allora non si può fare `s/$ricercavagna/xxx/`, perché il Perl penserà che ci si stia riferendo alla variabile `$ricercavagna`. Invece bisogna racchiudere il nome della variabile tra parentesi graffe ottenendo quindi:

```
$ricerca = "la";  
s/${ricerca}vagna/xxx/;
```

12.3 Trasformazione di caratteri

La funzione `tr` permette una traduzione (translation) carattere per carattere. L'espressione seguente sostituisce nella variabile `$frase` ogni "a" con "e", ogni "b" con "d", e ogni "c" con "f". Come valore di ritorno viene dato il numero di sostituzioni fatte.

```
$frase =~ tr/abc/edf/;
```

La maggior parte dei codici speciali relativi alle ER non hanno significato speciale quando usati nella funzione `tr`. Per esempio

```
$cont = ($frase =~ tr/*/*/);
```

conta il numero degli asterischi nella variabile `$frase` e memorizza il numero ottenuto nella variabile `$cont`.

In ogni caso il trattino - consente ancora di specificare una sequenza di caratteri consecutivi. Ecco come fare in modo che `$_` contenga solo maiuscole:

```
tr/a-z/A-Z/;
```

12.4 Esercizio

L'ultimo programma scritto dovrebbe contare le righe di un file che contengono una certa stringa. Modificarlo in modo che conti le righe con un qualsiasi carattere ripetuto. Modificarlo di nuovo in modo queste doppie lettere vengano messe in evidenza tra parentesi. Per esempio il programma potrebbe produrre una riga tipo la seguente tra le altre:

132 Pi(pp)o o(pp)ure Gia(nn)i.

Per fare un programma leggermente più interessante si potrebbe fare in modo che il programma prenda la stringa da cercare come primo argomento. Supponiamo che il programma si chiami `contarighe`. Se esso viene chiamato come

`./contarighe primo secondo etc`

allora gli argomenti vengono salvati nell'array `@ARGV`. Nell'esempio sopra `$ARGV[0]` sarebbe uguale a "primo", `$ARGV[1]` uguale a "secondo" e così via. Modificare il programma in modo che accetti una stringa come argomento e conti le righe contenenti tale stringa. Dovrebbe inoltre mettere tutte le occorrenze di quella stringa tra parentesi. Così

`./contarighe la`

potrebbe dare risultati di questo tipo:

231 Leggo al(la) luce del(la) (la)mpada.

Capitolo 13

Divisione stringhe

Una funzione molto utile del Perl è `split`, che spezza una stringa in pezzi mettendoli in un array. Per specificare cosa deve essere considerato come separatore si passa a questa funzione un'espressione regolare. Anche `split` agisce sulla variabile `$_`, se non si specifica diversamente.

Un esempio renderà tutto più chiaro:

```
$info = "Aldo:Riso:attore:via Vai, 3";  
@dati = split( /:/, $info );
```

Dopo queste due righe gli elementi dell'array `@dati` sono costituiti dagli spezzoni della stringa `$info`, come se avessimo fatto

```
@dati = ( "Aldo", "Riso", "attore", "via Vai, 3" );
```

Se la stringa di partenza viene posta in `$_`, allora basta fare

```
@dati = split( /:/ );
```

Se i campi della stringa in ingresso possono essere divisi da un separatore meno semplice, non ci sono comunque problemi, basta cambiare l'ER passata a `split`. Così le righe

```
$_ = "Rino:Nero::comico:::via Vai, 4";  
@dati = split( /:+/ );
```

hanno lo stesso effetto di

```
@dati = ( "Rino", "Nero", "comico", "via Vai, 4" );
```

mentre invece

```
$_ = "Rino:Nero::comico:::via Vai, 4";  
@dati = split( /:/ );
```

avrebbe avuto l'effetto di

```
@dati = ( "Rino", "Nero", "", "comico", "", "", "via Vai, 4" );
```

Una parola può essere divisa nei caratteri che la compongono, una frase in una serie di parole e un paragrafo in una sequenza di frasi:

```
@caratteri = split( //, $parola );  
@parole = split( /\s/, $frase );  
@frasi = split( /\./, $paragrafo );
```

Nel primo caso viene usato come separatore la stringa nulla, per cui l'array @caratteri conterrà una serie di stringhe di lunghezza unitaria come desiderato.

13.1 Esercizio

Uno strumento utile quando si processa linguaggio naturale è la concordanza. Questo permette di visualizzare una stringa specifica nel conteso che la circonda, ovunque essa compaia nel testo. Per esempio, un programma che cerca le concordanze per la stringa "per" potrebbe produrre un'uscita di questo tipo (notare che le occorrenze della stringa cercata sono state allineate al centro):

ngustia moltissime	persone. Ma cosa fare
one. Ma cosa fare,	per risolverlo? Eleme
azione delle gambe	per rallentare o prev
l'attività fisica	per tonificare la mus
ana è lo stile che	permette di avere una
la testa sollevata	per orientarsi in acq
essere facilissimo	per chi non ha il col

L'esercizio consiste proprio nello scrivere un programma di questo tipo. Qui alcuni consigli per partire:

- Leggere l'intero file in un array (questa tecnica ovviamente non può essere utilizzata in generale, perché il file potrebbe essere estremamente grande, ma adesso non ci preoccupiamo di questo). Ogni elemento dell'array conterrà una riga del file.
- Usare la funzione chop sull'array. Questo equivale a togliere l'ultimo carattere (in questo caso il newline) da ogni elemento dell'array.
- Sfruttare la stringa "per" come separatore per la funzione split (al posto di :). Sarà allora facile ottenere un array con tutte le stringhe comprese tra due successivi "per".
- Per ogni occorrenza di "per" scrivere la stringa che lo precede, poi "per" ed infine la stringa che lo segue.
- Si ricordi che l'ultimo elemento dell'array @arr ha indice \$#arr.

A questo punto ci siamo avvicinati allo scopo voluto, ma la stringa "per" non è ancora allineata al centro. Per mettere in ordine le stringhe conviene allora utilizzare la funzione substr. Ecco tre esempi del suo uso:

```
substr( "C'era una volta", 3, 4 ); # restituisce "ra u"  
substr( "C'era una volta", 7 ); # restituisce "na volta"  
substr( "C'era una volta", -7, 6 ); # restituisce "a volt"
```

Nel primo caso viene restituita la sottostringa di lunghezza 4 con inizio alla posizione 3. Si ricordi che il primo carattere ha indice 0. Nel secondo caso si vede come omettendo la lunghezza si ottenga una sottostringa che si estende fino alla fine della stringa di partenza. Infine l'ultimo esempio mostra come si possa specificare la posizione di inizio a partire dalla fine, usando un indice negativo. Viene ritornata la sottostringa lunga 6 caratteri che inizia al settimo carattere dalla fine.

Se si usa un indice negativo che si riferisce ad una posizione che precede l'inizio della stringa, allora il Perl non ritornerà nulla e potrebbe generare un messaggio di avvertimento. Per evitare questo, eventualmente si può aggiungere una lunga sequenza di spazi all'inizio della stringa sulla quale si sta operando. Ad esempio:

```
$pippo = " "x30 . $pippo;
```

Come si è visto nel capitolo sulle variabili scalari, l'operatore x serve per ripetere un numero specificato di volte una certa stringa, invece . è l'operatore di concatenazione.