

Michele Beltrame

# Corso di Perl

Questo corso a puntate è stato pubblicato sui numeri 75,76,77,78,79,80,81,82,83,84 (2000-2001) della rivista Dev ed è in questo sito (<http://www.perl.it>) per autorizzazione espressa del "Gruppo Editoriale Infomedia" (<http://www.infomedia.it>). È vietata ogni riproduzione.

© 2005-2007 Michele Beltrame. Michele Beltrame vive a Maniago, PN, è programmatore Perl e convinto sostenitore del movimento Open Source. Può essere raggiunto via e-mail all'indirizzo [arthas@perl.it](mailto:arthas@perl.it). © Perl Mongers Italia. Tutti i diritti riservati. Revisione 0.10

## Contatti

### Web

<http://www.perl.it>

### Mail

[info@perl.it](mailto:info@perl.it)

### Newsgroup

<news://it.comp.lang.perl>

## Indice

1. Introduzione - pag. 2
2. I tipi di dato e gli operatori - pag. 5
3. Array ed hash - pag. 11
4. Condizioni e iterazioni - pag. 17
5. Reference, subroutine, parametri - pag. 24
6. Funzioni e regular expression - pag. 31
7. Approfondimenti sulle regular expressions - pag. 38
8. Gestione dei file e dei programmi esterni - pag. 45
9. Le funzioni principali di Perl - pag. 52
10. Altre funzioni e moduli - pag. 58
11. Listati - pag. 64
12. Bibliografia - pag. 65

# 1. Introduzione

Con l'espansione di Internet, Perl si è progressivamente diffuso come linguaggio di *scripting*, cioè come linguaggio interpretato da usarsi per creare, in maniera semplice e veloce, applicazioni server-side, tra cui script CGI. In realtà Perl è molto più di questo, in quanto si tratta di un completo linguaggio di programmazione con cui si possono creare applicativi di ogni tipo, dai semplici script per la gestione del sistema a complesse applicazioni con tanto di sofisticata interfaccia grafica. Uno degli aspetti che ha portato al grande successo di Perl è la sua portabilità: lo stesso sorgente può infatti essere eseguito, praticamente senza alcuna modifica, su piattaforme che vanno da Linux a Windows a MacOS. Perl è attualmente utilizzato da un numero grandissimo di sviluppatori, e questo ha fatto sì che ora sia possibile trovare in giro qualsiasi libreria per il linguaggio, per l'interfacciamento con qualsiasi software o periferica, rendendolo quindi adatto per qualsiasi uso, al pari di linguaggi dal nome più blasonato quali C++ oppure Java.

## Compilato o interpretato?

Perl è tradizionalmente un linguaggio interpretato. Se dal punto di vista della velocità può rappresentare un problema, questo è stato invece uno dei punti di forza del linguaggio: il non dover compilare i sorgenti semplifica infatti notevolmente lo sviluppo in quanto si possono tranquillamente sviluppare i propri programmi sul proprio computer di casa (che ad esempio gira sotto Microsoft Windows o Linux) e poi caricare direttamente i sorgenti sul server di rete su cui devono girare (nel quale è magari installato FreeBSD o qualche altro sistema operativo). Altro vantaggio di un linguaggio interpretato è la semplicità con cui si può correggere un errore in un sorgente già caricato sul server: basta un client FTP ed un normale editor di testi ed il gioco è fatto.

Nonostante tutto, ad un certo punto si è presentata la necessità, per alcuni scopi che necessitano di una certa velocità di elaborazione, di compilare i sorgenti Perl in eseguibili nativi. Il primo a tentare di risolvere il problema è stato Malcom Beattie, il quale ha creato due moduli che si occupano l'uno di compilare il sorgente Perl in un *bytecode* semi-interpretato (qualcosa di simile ai file *.class* del Java) e l'altro di convertire il sorgente da Perl a C. La prima soluzione in effetti velocizza l'esecuzione del tutto, ma il programma generato rimane dipendente da troppe librerie "accessorie". Il secondo modulo crea invece un codice C poco piacevole da leggere ma facilmente compilabile con un qualsiasi compilatore quale ad esempio GCC: si può a questo punto scegliere se ottenere un eseguibile linkato dinamicamente (ed in questo caso bisognava portarsi dietro una libreria) oppure un gigantesco eseguibile linkato staticamente. Le soluzioni di Malcom sono quelle che ancora oggi si rivelano vincenti per la compilazione del Perl: il problema era che solo una parte dei programmi Perl potevano essere dati in pasto al suo software, perennemente in fase di alpha testing. Oggi esistono alcuni compilatori commerciali di buon livello quali Perl2Exe e PerlApp [Ed ora anche uno Open Source: PAR, NdR].

## I vari kit di sviluppo

Esistono ormai varie distribuzioni di Perl, per molte piattaforme. La più famosa è naturalmente l'originale, quella creata da Larry Wall, che funziona praticamente su tutti i sistemi Unix esistenti e, nelle versioni più recenti, anche su Windows. La versione attuale è la 5.8.2, in costante aggiornamento. Parallelamente è partito un progetto di riscrittura dell'intero pacchetto, con

ridefinizione radicale del linguaggio in sé, che dovrebbe portare a Perl 6. Di fatto sotto Unix non serve in generale preoccuparsi di procurarsi il linguaggio, in quanto viene incluso nell'installazione base della quasi totalità dei sistemi. Esistono poi le distribuzioni per Windows, la più nota delle quali è ActivePerl, alla quale collabora la stessa Microsoft, e che implementa più o meno tutte le caratteristiche del Perl originale. Recentemente sono inoltre comparsi i primi tool di sviluppo grafici. La stessa ActiveState, creatrice di ActivePerl e PerlApp, propone un IDE, Komodo, che supporta anche un certo numero di altri linguaggi ed include un debugger.

## Risorse in rete

Il maggiore supporto al linguaggio arriva da Internet: sono disponibili infatti risorse di ogni tipo. Il sito "istituzionale" è <http://www.perl.com>, creato dal Perl guru Tom Christiansen ed attualmente gestito da O'Reilly & Associates: qui si trovano i link a tutto quello che serve per iniziare a programmare, nonché tutte le ultime novità sul linguaggio. Altro sito leggendario è il CPAN (Comprehensive Perl Archive Network, <http://www.cpan.org>), che raccoglie praticamente tutti i moduli (librerie) esistenti ed ha mirror in tutto il mondo. Molto interessante è anche Perl Mongers (<http://www.perl.org>), contenente vari link interessanti ed è utilizzabile ad esempio per cercare un posto di lavoro che coinvolga la conoscenza del Perl. Un valido aiuto, un punto di incontro dove scambiarsi consigli sul Perl oppure semplicemente fare quattro chiacchiere è rappresentato dagli user group di Perl Mongers (<http://www.pm.org>), moltissimi e sparsi in tutto il mondo: dal sito centrale è possibile accedere a quelli relativi alle specifiche aree geografiche (in Italia ci sono Bologna.pm, Nordest.pm, Pisa.pm, Roma.pm, Salerno.pm e Torino.pm). Su quest'ultimo sito inoltre possibile acquistare un po' di abbigliamento "perloso", nonché registrarsi in modo da essere reperibili da altri programmatori che vivono nell'area dove si risiede. PerlMonks (<http://www.perlmonks.com>) è una grandiosa risorsa dove chiedere aiuto, cercare informazioni e scambiare opinioni; è frequentato dai più noti programmatori Perl a livello mondiale. Il punto di riferimento in italiano è invece Perl.It (<http://www.perl.it>). Un corso online e molti script da cui trarre spunto sono presenti su <http://www.html.it>. Ottime risorse sono inoltre i newsgroup: in Italia è stato attivato `it.comp.lang.perl`, in cui c'è poco "rumore di fondo" ed un po' di gente disponibile a fornire preziosi consigli.

## Un primo sguardo al linguaggio

A livello sintattico Perl, come più recentemente anche Java, ha ereditato quasi tutto da C. Di seguito è riportato il classico sorgente che scrive "Ciao, Mondo!":

```
#!/usr/local/bin/perl5 -w  
  
print "Ciao, Mondo!\n";
```

La prima riga è di vitale importanza solo in ambiente Unix, in quanto si occupa di invocare l'interprete quando lo script viene eseguito chiamandolo dalla linea di comando semplicemente digitandone il nome (altrimenti sarebbe stato necessario digitare `perl -w nomeprog.pl` per garantirne l'esecuzione). Sotto Windows la presenza di tale riga non dà alcun fastidio, in quanto il cancelletto indica che ogni carattere da quel momento in poi e fino alla fine della riga deve essere considerato commento, e quindi non interpretato. È invece importante, se si usa Windows, garantire l'associazione tra i file aventi estensione `.pl` (e magari `.cgi`) e l'interprete: questo è

configurato in automatico da alcune distribuzioni provviste di installazione, come ActivePerl, mentre per altre bisogna provvedere manualmente. Guardando bene la prima riga si nota anche il flag `-w`: questo indica all'interprete di visualizzare tutti i *warning* (vedremo nelle prossime puntate di che cosa si tratta di preciso), ed il suo utilizzo è sempre raccomandato, ma sfortunatamente non abbastanza diffuso. L'unica riga vera e propria di codice è quella dell'istruzione *print*, che si occupa di stampare a video la stringa contenuta tra i doppi apici. Il `\n` con cui termina la stringa indica a *print* di aggiungere un line feed (cioè di andare a capo) a fine riga. Qualche lettore che si diletta con il linguaggio C si sarà accorto che non è necessario includere alcuna libreria per l'utilizzo delle funzioni standard del linguaggio: l'interprete Perl include infatti già tutte le funzioni di base, e moltissime sono disponibili nei moduli presenti direttamente nelle distribuzioni, i quali vanno però inclusi con l'apposita istruzione, come vedremo in futuro. Di fatto la maggior parte delle operazioni può essere compiuta senza l'ausilio di librerie esterne ed anche in più modi: questo è un aspetto caratteristico del Perl, infatti una tipica frase di chi usa questo linguaggio è "c'è più di un modo per fare la stessa cosa".

## Conclusioni

Con questa prima lezione si è più che altro cercato di far capire cos'è Perl, e di introdurre la filosofia che c'è dietro il linguaggio: questa di fatto si comprenderà a poco a poco, ed il perché dei luoghi comuni sulla pigrizia dei programmatori Perl sarà chiaro nelle prossime lezioni.

Gli utilizzatori del Perl costituiscono di fatto un vera e propria comunità (come si noterà visitando il sito degli user group dei Perl Mongers, descritto precedentemente): da anni negli Stati Uniti, e più recentemente anche in tutto il resto del mondo, meeting, conferenze e cene sono piuttosto frequenti. L'interprete Perl è disponibile con licenza open source (Artistic o GPL, a scelta): gli utilizzatori hanno ereditato tale filosofia, facendo del confronto con gli altri l'aspetto più importante del loro lavoro.

## 2. I tipi di dato e gli operatori

**Perl offre svariati tipi di dato, la cui gestione è molto intuitiva. Per quanto riguarda gli operatori, essi sono veramente tanti: in alcuni casi è compito del programmatore scegliere quelli che ritiene più comodi ed eleganti.**

### Introduzione

La prima cosa che si apprende quando si studia un linguaggio di programmazione è, naturalmente dopo il classico programma che scrive Ciao, Mondo!, la gestione dei dati: ciò significa capire quali tipi di dato il linguaggio mette a disposizione, e quali operatori si devono utilizzare per gestire tali dati. In Perl la vita è abbastanza facile, in quanto ad esempio non servirebbe dichiarare le variabili. Il condizionale è d'obbligo, poiché è raccomandabile forzare il linguaggio a richiederne la dichiarazione: questo toglie un po' di libertà al programmatore, ma permette di creare programmi più ordinati e rende il debugging molto più rapido. La dichiarazione delle variabili verrà comunque introdotta nelle lezioni successive.

### Scalari

Il glossario della prima edizione di *Programming Perl* indica uno *scalare* come "un valore semplice, come un numero o una stringa. Oppure, qualcuno che scala le montagne". Nell'edizione successiva del testo, l'unica che è ora possibile trovare nelle librerie, questo umorismo è purtroppo sparito dal glossario, e ne è rimasta solo la parte "seria". In ogni caso, come da definizione, uno scalare è il tipo di dato fondamentale: stringhe di qualunque tipo e numeri, sia interi che decimali, sono valori scalari. Assegnare un valore numerico ad uno scalare è piuttosto semplice:

```
$numero = 1974;
```

Il dollaro è il carattere che identifica uno scalare, e va sempre posto prima del nome della variabile. L'operatore di assegnazione è il segno di uguale (=). Per vedere il contenuto della variabile creata basta scrivere:

```
print "$numero\n";
```

Un numero può essere assegnato utilizzando varie notazioni: ad esempio è disponibile *\$mynum=287.32* per i decimali, *\$mynum = 7.5E15* per gli esponenziali, *\$mynum = 0xffff* per gli esadecimali oppure *\$mynum = 0377* per gli ottali. Per quanto riguarda le stringhe, il discorso è notevolmente più articolato. Una stringa può essere inizializzata nel seguente modo:

```
$stringa = 'Ciao, Mondo\n';
```

oppure così:

```
$stringa = "Ciao, Mondo\n";
```

La differenza tra l'uso dei singoli apici e quello dei doppi apici è immediatamente comprensibile se si stampa a video la stringa creata: quella inizializzata utilizzando i singoli apici apparirà come *Ciao*,

*Mondo*`\n`, mentre la seconda sarà visualizzata come *Ciao, Mondo* seguita da un newline (passaggio alla linea successiva). La differenza è quindi nell'interpolazione: per le stringhe inizializzate con i doppi apici Perl interpreta e sostituisce i caratteri a seguito di un backslash (`\`), mentre per quelle inizializzate con i singoli apici ciò non avviene (eccezion fatta per `\\` e `'`, indispensabili per inserire il backslash e l'apice nella stringa). Se si inizializza una variabile utilizzando i doppi apici, anche i nomi di altre variabili inseriti all'interno di essa vengono interpolati, quindi ad esempio:

```
$stringa1 = "Ciao, ";  
$stringa2 = "$stringa1 Mondo!";
```

produrrà una variabile `$stringa2` contenente il classico testo *Ciao, Mondo!*. Va notato che quanto detto vale anche per le funzioni che accettano stringhe come parametri, quali `print`: esse accettano sia singoli che doppi apici, e le stesse regole sono applicate. Oltre a `\n`, molti altri caratteri possono essere utilizzati a seguito di un backslash per rappresentare stringhe speciali: una lista di essi è visibile in **tabella 1**.

### Carattere Significato

<code>\n</code>	Newline
<code>\r</code>	Carriage return (sotto Windows le righe finiscono con <code>\r\n</code> , sotto Unix solo con <code>\n</code> )
<code>\t</code>	Tabulazione orizzontale
<code>\f</code>	Form feed
<code>\b</code>	Backspace
<code>\a</code>	Allarme (causa l'emissione di un suono dallo speaker del computer)
<code>\e</code>	Carattere ESCAPE
<code>\033</code>	Qualsiasi carattere, in notazione ottale (in questo caso è ESCAPE)
<code>\x7f</code>	Qualsiasi carattere, in notazione esadecimale (in questo caso è ESCAPE)
<code>\cS</code>	Qualsiasi carattere di controllo (in questo caso CTRL-S)
<code>\\</code>	Backslash
<code>\"</code>	Apici doppi
<code>\u</code>	Fa sì che il carattere successivo a <code>\u</code> sia convertito in maiuscolo
<code>\l</code>	Fa sì che il carattere successivo a <code>\l</code> sia convertito in minuscolo
<code>\U</code>	Fa sì che tutti i caratteri successivi a <code>\U</code> siano convertiti in maiuscolo
<code>\L</code>	Fa sì che tutti i caratteri successivi a <code>\L</code> siano convertiti in minuscolo
<code>\Q</code>	Fa sì che tutti i caratteri successivi e non alfanumerici siano automaticamente preceduti da backslash
<code>\E</code>	Termina l'effetto di <code>\U</code> , <code>\L</code> oppure <code>\Q</code>

### Operazioni semplici con i numeri

Gli operatori disponibili per le operazioni con gli scalari a valore numerico sono quelli classici. Ad esempio:

```
$c = $a + $b;
```

memorizzerà in `$c` la somma dei valori di `$a` e `$b`. Altre possibilità sono la sottrazione (operatore `-`), la moltiplicazione (operatore `*`), la divisione (operatore `/`), il resto della divisione (modulo, operatore `%`) e l'esponenziale (operatore `**`). È possibile utilizzare una forma abbreviata per

sommare un numero fisso ad una variabile. Ad esempio, al posto  $\$a = \$a * 2$ , si può scrivere:

```
\$a *= 2;
```

Quando si desidera sottrarre o aggiungere una sola unità ad una variabile a contenuto numerico esiste, come in C, una forma ancora più abbreviata, e cioè la semplice istruzione  $\$a++$  oppure  $++\$a$  per quanto riguarda l'incremento, e  $\$a--$  oppure  $--\$a$  per quanto riguarda il decremento. Cosa cambia tra anteporre e postporre l'operatore? Entrambi hanno lo stesso effetto (e cioè di incrementare o decrementare una variabile), ma analizzando il contesto si nota una differenza fondamentale. Vediamo un esempio:

```
\$a = 1;
\$b = 1;
print \$a++, "\n";
print ++\$b, "\n";
```

L'output è rappresentato da due righe, una con scritto 1 e l'altra con scritto 2, benché alla fine tutte e due le variabili assumano valore 2. La differenza tra la prima e la seconda sintassi risiede quindi nel momento in cui avviene l'incremento: se l'operatore è posto dopo il nome della variabile, essa viene prima utilizzata nel contesto (la funzione print in questo caso) e poi incrementata; se l'operatore è invece anteposto al nome della variabile, essa viene prima incrementata e poi utilizzata nel contesto.

## Operazioni semplici con le stringhe

Come per i numeri, anche per le stringhe il Perl fornisce una serie di operatori utili alla loro gestione. Per concatenare due stringhe si procede come da seguente esempio:

```
\$a = "prova!";
\$b = "questa è una " . \$a;
```

L'operatore da usare è quindi il punto. In realtà, utilizzando i doppi apici, l'operatore di concatenazione è del tutto superfluo, in quanto sarebbe sufficiente la seguente forma:

```
\$b = "questa è una \$a";
```

Con questo non si vuole assolutamente dire che tale operatore è del tutto superfluo: esso è infatti utile quando ad esempio si vuole inserire in una stringa il valore restituito da una funzione, come vedremo nelle successive lezioni. L'operatore risulta inoltre utile se una variabile va inserita all'interno di un testo senza essere separata da spazi. Ad esempio potremmo scrivere quanto segue:

```
\$a = "pr";
\$b = "questa è una ".$a."ova";
```

In  $\$b$  avremo il solito testo questa è una prova. Non avrebbe invece funzionato se come seconda riga avessimo usato:

```
\$b = "questa è una $aova";
```

in quanto il nome di variabile preso in considerazione sarebbe stato *\$aova* e non *\$a*. In tipico stile Perl c'è naturalmente un altro metodo per ottenere lo stesso scopo. Esso prevede l'uso delle parentesi graffe:

```
$b = "questa è una ${a}ova";
```

Sorprendentemente, le stringhe si possono anche moltiplicare:

```
$a = "abc";  
$b = "abc" x 3;
```

Il risultato è la stringa *\$a* ripetuta tre volte, e cioè *abcabcabc*. Un importantissimo operatore per le stringhe è poi *=~*, cioè quello relativo al *pattern matching*: ad esso ed alle famose *regular expression* verrà dedicata un'intera lezione.

## Il costrutto if, gli operatori logici ed i confronti tra stringhe e numeri

Comprendere il funzionamento della programmazione condizionale è indispensabile: al verificarsi di una determinata condizione viene eseguito un certo codice, al verificarsi di un'altra altro codice, e via dicendo. Il costrutto principe della programmazione condizionale è *if-elsif-else*, di cui vediamo subito un esempio:

```
if ($a == 1) { print "vale 1!"; }  
elsif ($a >= 2) { print "vale 2 o più!"; }  
else { print "vale altro!"; }
```

Il codice posto tra parentesi graffe viene eseguito solo se l'espressione immediatamente precedente restituisce un valore vero (in questo caso se la comparazione è esatta); se il valore *booleano* restituito è *false* nulla viene invece eseguito. In questo esempio confrontato il valore della variabile *\$a*: se è uguale a *1* viene stampato a video il testo *vale 1!*, se invece è maggiore o uguale a *2* viene stampato *vale 2 o più!*, mentre in tutti gli altri casi viene stampato *vale altro!* L'istruzione *elsif*, a differenza di *if*, viene eseguita solo se l'*if* precedente ha avuto esito negativo; *else* a sua volta viene eseguita solo in caso di esito negativo dei confronti avvenuti in precedenza. Non è assolutamente indispensabile che ad un *if* faccia seguito un *elsif* oppure un *else*, ma questi ultimi due richiedono assolutamente la presenza del primo.

Tutti gli operatori disponibili per il confronto dei numeri e delle stringhe sono visibili in **tabella 2**.

Comparazione	Operatore numerico	Operatore stringa	Valore restituito
Uguaglianza	==	eq	Vero se \$a è uguale a \$b, falso altrimenti
Disuguaglianza	!=	ne	Vero se \$a è diverso da \$b, falso altrimenti
Minoranza stretta	<	lt	Vero se \$a è minore di \$b
Maggioranza stretta	>	gt	Vero se \$a è maggiore di \$b
Minoranza	<=	le	Vero se \$a è minore o uguale a \$b
Maggioranza	>=	ge	Vero se \$a è maggiore o uguale a \$b
Comparazione	<=>	cmp	-1 se \$b è maggiore, 0 se \$a è uguale a \$b, 1 se \$a è maggiore

Con una sola istruzione `if` oppure `elsif` si possono eseguire più comparazioni, utilizzando gli operatori logici per collegarle l'una all'altra: essi sono fondamentalmente tre: `||`, `&&` e `!`. Di questi esistono anche gli equivalenti a priorità più bassa, che sono rispettivamente `or`, `and` e `not`. La questione della priorità tra gli operatori in Perl è abbastanza delicata, e verrà approfondita in futuro: per ora il consiglio è quello di includere le varie espressioni tra parentesi quando si è nel dubbio. Tornando alla funzione dei vari operatori, analizziamo i seguenti esempi:

```
if ( ($a == 1) && ($b == 2) ) { print "OK!"; }
if ( ($a == 1) || ($b == 2) ) { print "OK!"; }
if ( !($a == 1) ) { print "OK!"; }
```

Nel primo caso il codice condizionale verrà eseguito solo se si verificano entrambe le condizioni: `$a` deve valere 1 e `$b` deve valere 2. Nel secondo caso per garantire l'esecuzione del codice condizionale è sufficiente che una delle due condizioni si verifichi, oppure entrambe: solo il non verificarsi di alcuna delle due provocherà la mancata esecuzione di `print`. La terza riga provoca invece l'esecuzione della parte condizionale solo se la condizione è falsa: il `!` (not) ribalta infatti il risultato dell'espressione logica, quindi `$a` deve essere di fatto diverso da 1. Si poteva anche scrivere:

```
if ($a != 1) { print "OK!"; }
```

Il Perl mostra tuttavia la sua eleganza mettendo anche a disposizione un costrutto che, pur avendo un funzionamento del tutto analogo a `if`, ribalta automaticamente il significato logico dell'espressione. La riga precedente può quindi essere scritta anche così:

```
unless ($a == 1) { print "OK!"; }
```

## Stringa o numero? Una questione di contesto

In Perl il problema della conversione di una stringa in numero e viceversa è risolto dall'interprete: esso converte automaticamente gli scalari tra un tipo e l'altro a seconda del contesto in cui essi sono inseriti. Poniamo di definire due variabili come segue:

```
$a = "100";
$b = 200;
```

Il primo scalare è definito come stringa, il secondo come intero. Possiamo tuttavia tranquillamente effettuare una somma:

```
$c = $a + $b;
```

ed ottenere correttamente *300* come risultato, grazie alla conversione automatica della stringa in valore numerico. Allo stesso modo funzionerà:

```
$c = $a . $b;
```

Il risultato in questo caso è la stringa *100200*, grazie alla conversione automatica in stringa del numero contenuto in *\$b*, effettuata a causa della presenza dell'operatore di concatenazione. È importante notare che queste conversioni vengono effettuate automaticamente anche quando si

effettuano dei confronti, come ad esempio avviene all'interno di un'istruzione *if*. Anche in questo caso è l'operatore usato a decidere il contesto: se ad esempio si usa `==` verrà testata l'uguaglianza tra due numeri, se si usa `eq` verrà invece testata l'uguaglianza tra due stringhe.

## **Conclusioni**

Abbiamo visto ed analizzato i tipi base di dato in Perl. Come si nota, non è mai necessario indicare il tipo di variabile, come ad esempio va fatto in C: è l'interprete a gestire il tutto, e ad occuparsi delle conversioni qualora necessario.

# 3. Array ed hash

**Gli array e gli hash sono le strutture dati di base del Perl, e permettono di raggruppare valori scalari in vari modi. Utilizzandoli è possibile gestire anche strutture più avanzate, che in altri linguaggi avrebbero richiesto codice specifico.**

## Introduzione

Una delle prime cose di cui tener conto quando si inizia a progettare un programma è l'organizzazione dei propri dati. Le variabili scalari possono andare bene per applicazioni molto semplici, ma quando ad esempio c'è una lista di nomi da memorizzare è necessario ricorrere a strutture più complesse. Perl mette anzitutto a disposizione, come praticamente tutti i linguaggi di programmazione, gli array, cioè collezioni di valori a ciascuno dei quali è assegnato un indice numerico che permette di leggerli o modificarli. Sono poi già inclusi nel linguaggio, e questa è una peculiarità del Perl, gli hash, conosciuti anche come dizionari o array associativi: questi sono sempre delle collezioni di valori a ciascuno dei quali è assegnata però una stringa (chiamata chiave) anziché un numero incrementale. Vediamo ora nel dettaglio le due strutture.

## Array

Un array, chiamato a volte impropriamente lista, è un gruppo di valori scalari nel quale è mantenuto un indice sequenziale che permette al programmatore di memorizzare un valore in una determinata posizione e di leggerlo o modificarlo in un secondo momento. Poniamo di dover gestire nel nostro programma quattro nomi di persona; potremmo memorizzarli in altrettante variabili scalari:

```
$nome1 = 'Michele';  
$nome2 = 'Elena';  
$nome3 = 'Luca';  
$nome4 = 'Elisa';
```

Questo sistema può anche andare bene se i nomi non sono moltissimi, ma in realtà si rivela già poco elegante se ne stiamo trattando più di due. Un array permette una gestione molto più comoda del tutto, e può essere inizializzato come segue:

```
@nomi = ('Michele', 'Elena', 'Luca', 'Elisa');
```

L'inizializzazione è un'operazione del tutto facoltativa, in quanto si possono direttamente inserire gli elementi nella posizione desiderata secondo la sintassi che vedremo tra poche righe. Se le stringhe prevedono solo caratteri alfanumerici è possibile anche utilizzare la cosiddetta *quoted syntax*:

```
@nomi = qw(Michele Elena Luca Elisa);
```

che permette di evitare di dover sempre scrivere gli apici. Questa soluzione non effettua tuttavia alcuna interpolazione, quindi se per inizializzare le stringhe utilizzate gli apici doppi poiché dovete inserire qualche carattere particolare o variabile, dovrete utilizzare *qq* in luogo di *qw*.

I due frammenti di codice appena visti creano un array di nome *nomi* con quattro elementi.

L'identificatore che contraddistingue gli array è, diversamente dagli scalari al cui nome è anteposto il dollaro, la chiocciolina (@). A questo punto l'array può essere gestito in molti modi: se ad esempio vogliamo stamparne tutto il contenuto possiamo scrivere:

```
print @nomi;
```

Questo causerà la stampa dei vari elementi, nell'ordine in cui erano stati inseriti, senza alcun separatore; nel nostro caso l'output sarà quindi:

```
MicheleElenaLucaElisa
```

La cosa più utile è naturalmente gestire i vari elementi singolarmente. Ciò è molto semplice, in quanto ogni elemento può essere trattato come uno scalare. Quindi funzionano benissimo le seguenti istruzioni:

```
print $nomi[0];  
$nomi[2] = 'Daniele';
```

La prima riga stampa il primo elemento dell'array (in Perl, come in molti altri linguaggi, l'indice di partenza è 0 e non 1), mentre la seconda cambia il valore del terzo elemento da *Luca* a *Daniele*. Come si nota, quando si gestiscono i singoli elementi non si antepone più la chiocciolina al nome dell'array, ma il dollaro: questo è perfettamente coerente, in quanto come detto ogni elemento viene trattato come uno scalare. Le parentesi quadre permettono di specificare l'indice (cioè la posizione) in cui è memorizzato l'elemento a cui interessa accedere. È anche possibile accedere a porzioni di array:

```
print @nomi[1,3];  
print @nomi[1..3];
```

La prima istruzione stampa solo gli elementi di indice 1 e 3 dell'array: si può specificare un numero arbitrario di elementi separati da virgola. La seconda istruzione permette invece di stampare un intervallo, in questo caso quello che va dall'elemento 1 all'elemento 3, estremi compresi. Utilizzando le sintassi sopra descritte si possono anche creare nuovi array composti da parti o da tutto l'array originale, come nei seguenti esempi:

```
@nomi2 = @nomi;           # Copia tutto @nomi in @nomi2  
@nomi3 = @nomi[1,3];     # Copia gli elementi 1 e 3 in @nomi3  
                        # (in cui diventano 0 e 1)  
@nomi4 = @nomi[1..3];    # Copia gli elementi 1, 2 e 3 in @nomi4  
                        # (in cui diventano 0, 1 e 2)
```

In molti casi è utile conoscere il numero di elementi di un array; è possibile procedere come segue:

```
print scalar(@nomi);
```

In contesto scalare una variabile di tipo array indica il numero di elementi in essa contenuti. La funzione *print()*, come si è visto prima, non cambia il contesto in scalare, e bisogna quindi ricorrere alla funzione *scalar()*. In altri ambiti il contesto viene automaticamente cambiato, come nel seguente esempio:

```
if (@nomi > 2) { print "L'array ha più di 2 elementi!"; }
```

Il frammento sopra riportato funziona correttamente poiché l'interprete si rende conto che il contesto dell'espressione *if* è scalare, e quindi modifica di conseguenza il significato di *@nomi*. Un'altra sintassi che permette di conoscere il numero di elementi ma che non risente di problemi di contesto è la seguente:

```
print $#nomi;
```

È tuttavia necessario prestare attenzione: questa sintassi in realtà restituisce *l'indice dell'ultimo elemento* e non *il numero di elementi* dell'array. Nel nostro caso *scalar(@nomi)* visualizza 4, mentre *\$#nomi* visualizza 3, poiché gli array in Perl hanno indice iniziale 0: quindi *scalar(@nomi) = \$#nomi+1*.

In Perl il ridimensionamento degli array avviene in maniera automatica: un array viene allungato o accorciato dinamicamente quando si inseriscono e tolgono degli elementi. Se ad esempio, prendendo sempre in considerazione l'array descritto sopra, diamo il comando:

```
$nomi[10] = 'Alessia';
```

viene automaticamente allocata memoria per tutti gli elementi fino all'undicesimo (quello cioè con indice 10). Allo stesso modo possiamo semplicemente accorciarne la dimensione con:

```
$#nomi = 2;
```

che imposta l'indice più alto a due, tagliando quindi fuori tutti gli elementi successivi.

## Code e stack con gli array

Esistono delle comode funzioni che permettono di gestire gli elementi collocati agli estremi (testa e coda) degli array, e che permettono quindi di realizzare facilmente strutture come *code* e *stack* (detti anche pile). La testa di un array è costituita dall'elemento di indice più basso (cioè 0); è possibile aggiungere un elemento alla testa con la seguente funzione:

```
unshift @nomi, 'Cristian';
```

Questo causa l'inserimento del nome *Cristian* nell'array: esso assume indice 0, e tutti gli altri elementi si trovano di conseguenza il loro indice incrementato di una unità. Per rimuovere un elemento dalla testa si può utilizzare:

```
$primo = shift @nomi;
```

L'elemento rimosso è restituito e può essere memorizzato in una variabile: ancora non abbiamo visto le funzioni (che analizzeremo in una delle prossime puntate), quindi il meccanismo di restituzione dei valori verrà spiegato in tale sede. Dopo l'operazione, eventuali altri elementi contenuti nell'array si trovano chiaramente il loro indice decrementato di una unità. La gestione della coda (elemento dell'array con indice più alto) segue principi identici. Il comando per inserire un elemento alla fine dell'array è:

```
push @nomi, 'Cristian';
```

mentre quello per rimuoverlo è:

```
$primo = pop @nomi;
```

Le funzioni *push()* e *unshift()* possono in realtà inserire più di un elemento in coda o in testa all'array: è sufficiente passare come secondo parametro alla funzione un altro array, ad esempio:

```
push @nomi, @nuovinomini;  
unshift @nomi, @nuovinomini;
```

Vediamo brevemente come utilizzare questi comandi per implementare code e stack. Anzitutto, una coda è una struttura dati in cui viene inserito un elemento alla volta, ed uno viene estratto. Esse sono anche chiamate strutture *FIFO (first in first out)*, in quanto il primo elemento inserito è il primo ad uscire: in poche parole, la coda restituisce gli elementi nello stesso ordine in cui sono stati inseriti (un esempio è costituito dalle automobili al casello autostradale: la prima che arriva è la prima a passare). Appare a questo punto ovvio che *push()* e *shift()* possono servire perfettamente al caso: la prima viene utilizzata per aggiungere un elemento in coda all'array, e la seconda per prelevarne uno dalla testa. Invertendo i concetti di testa e coda dell'array, è possibile implementare una coda anche utilizzando *unshift()* e *pop()* ma, per ragioni di efficienza dell'interprete Perl, è consigliato l'uso dei primi due.

Uno stack è invece una struttura *LIFO (last in first out)*, cioè una struttura dati in cui il primo elemento ad uscire è l'ultimo che è entrato. L'esempio pratico che dà il nome alla struttura (la cui traduzione è, come detto sopra, *pila*) è appunto la pila dei piatti lavati in una cucina: l'ultimo piatto lavato va a finire in cima, ed è quindi il primo ad essere riutilizzato. L'implementazione è anche in questo caso piuttosto semplice, e prevede l'uso di *push()* per inserire gli elementi e di *pop()* per estrarli in ordine inverso a quello di inserimento.

## Splice

Una delle funzioni più potenti messe a disposizione dal Perl per la manipolazione degli array è *splice()*. Essa è paradossalmente anche la meno utilizzata, in quanto per l'uso comune sono più che sufficienti le quattro funzioni sopra descritte (*push()*, *pop()*, *unshift()*, *shift()*), che di fatto rappresentano dei "casi particolari" di *splice()*. Quest'ultima può infatti fare le veci di tutte le precedenti: le equivalenze sono visibili in **tabella 1**.

Funzione	Equivalente <i>splice()</i>
<code>push (@arr, @newarr);</code>	<code>splice (@arr, \$#newarr+1, 0, @newarr);</code>
<code>\$elem = pop (@arr);</code>	<code>\$elem = splice (@arr, -1);</code>
<code>unshift (@arr, @newarr);</code>	<code>splice (@arr, 0, 0, @newarr);</code>
<code>\$elem = shift (@arr);</code>	<code>\$elem = splice (@arr, 0, 1);</code>
<code>\$arr[\$i] = \$elem;</code>	<code>splice (@arr, \$i, 1, \$elem);</code>

Di fatto *splice()* rimuove tutti oppure un certo numero di elementi da un array a partire da un certa posizione, rimpiazzandoli eventualmente con altri. Di seguito ne sono riportati alcuni possibili utilizzi:

```
# Rimuove gli elementi di indice da 2 a 4  
splice (@nomi, 2, 2);
```

```

# Fa lo stesso, con sintassi diversa
# (parte dal fondo, ma il risultato è il medesimo)
splice (@nomi, 4, -2);

# Rimuove tutti gli elementi con indice > 1
splice (@nomi, 2);

# Rimuove gli ultimi 3 elementi
splice (@nomi, -3);

# Sostituisce gli elementi do indice da 2 a 4
# con un nuovo array
splice (@nomi, 2, 4, @nuovonomi);

```

In quest'ultimo caso non è necessario che *@nuovonomi* sia esattamente di tre elementi: se è più lungo o più corto l'array viene automaticamente ridimensionato.

## Altre funzioni utili

Analizziamo in sintesi altre due funzioni utili nella gestione degli array, *reverse()* e *sort()*:

```

@invnomi = reverse @nomi;
@ordnomi = sort @nomi;

```

La prima inverte l'ordine degli elementi dell'array *@nomi*, memorizzando il risultato dell'operazione in *@invnomi*. La seconda salva invece in *@ordnomi* una versione ordinata di *@nomi*.

L'ordinamento utilizzato di default è quello alfabetico, ma è possibile cambiarlo passando una propria funzione di ordinamento: la cosa verrà discussa più in dettaglio quando si parlerà di funzioni, per ora vediamo solo un esempio di come si può ordinare un array contenente valori numerici:

```

@ordnumerici = sort {$a <=> $b} @numerici;

```

## Hash

Un *hash*, detto anche *array associativo*, è una struttura simile ad un array: al posto degli indici sono però utilizzate delle stringhe libere, dette *chiavi*, che permettono di leggere o scrivere il *valore* ad esse associato. L'inizializzazione, anche in questo caso del tutto facoltativa, avviene come segue:

```

%articolo = (
  'codice'      => 22,
  'nome'        => 'Cobra',
  'descrizione' => 'Coltello da collezione',
  'prezzo'      => 110000
);

```

L'identificatore degli hash è il segno di percentuale (%). Anche in questo caso, come per gli array, è possibile utilizzare la *quoted syntax* per assegnare i valori: sia virgole che frecce possono essere sostituite da spazi entro l'operatore *qw*. Di fatto la freccia (=>) ha per l'interprete lo stesso significato della virgola, ma il suo uso rende il listato più comprensibile. Abbiamo ora creato un hash di quattro elementi: due scalari a valore numerico (*codice* e *prezzo*) e due stringhe (*nome* e

*descrizione*). La sintassi per la gestione degli elementi di un hash è simile a quella che si usa per gli array. Per visualizzare un valore si scrive ad esempio:

```
print $articolo{'codice'};
```

Per impostarlo invece:

```
$articolo{'codice'} = 15;
```

Si può anche stampare l'intero hash con:

```
print %articolo;
```

L'output sarà costituito da ciascuna chiave seguita dal rispettivo valore, senza alcun separatore.

La funzione *reverse()* inverte le chiavi con i valori, assumendo che questi ultimi siano unici. La funzione *sort()* non deve invece essere applicata, in quanto un hash è per definizione non ordinato ed il suo uso porterebbe a risultati indesiderati. È possibile ottenere un array contenente tutte le chiavi oppure tutti i valori di un hash, tramite le funzioni *keys()* e *values()*:

```
@k = keys %articolo;  
@v = values %articolo;
```

Con il tempo ci si renderà conto che gli hash sono utilissimi in molti casi, ed il fatto che il Perl già li fornisca a livello di linguaggio è una comodità non da poco.

Array ed hash possono anche essere combinati tra loro per creare array multidimensionali, hash multidimensionali, array di hash e hash di array: per questo scopo è però indispensabile comprendere l'uso delle *reference*, che verranno spiegate più avanti.

## Conclusioni

In questa lezione sono state analizzate le due strutture dati fondamentali del Perl, a partire dalle quali se ne possono costruire molte altre, a seconda delle proprie esigenze. Alcuni dei concetti esposti possono apparire un po' nebulosi; essi saranno tuttavia più chiari una volta studiate funzioni e *reference*.

# 4. Condizioni e iterazioni

**In questa lezione verrà approfondita la programmazione condizionale mediante l'introduzione di costrutti alternativi al classico if. Verranno successivamente illustrare le strutture iterative fondamentali di Perl: for, while, until e foreach.**

## Introduzione

Nella scorsa lezione sono stati descritti i tipi di dato messi a disposizione da Perl: dalle semplici variabili scalari agli array agli hash. Quanto visto è sufficiente per scrivere qualche semplice programmino. Per scrivere qualcosa di realmente utile è tuttavia necessaria anche la conoscenza delle strutture iterative, quei costrutti che permettono cioè di creare del codice la cui esecuzione è ripetuta un numero di volte fisso oppure determinato da una o più condizioni. Prima di spiegare questi è tuttavia opportuno un approfondimento sulla programmazione condizionale: benché l'istruzione if sia sufficiente per un completo controllo del flusso del codice, essa risulta in molti casi essere poco pratica oppure poco elegante.

## Eleganza condizionale

Perl mette a disposizione svariate sintassi per quanto riguarda la programmazione condizionale: queste permettono di scrivere un codice che assomigli di più alla lingua parlata, il che significa rendere il codice lo più leggibile o completamente incomprensibile, a seconda del criterio con cui si utilizzano tali strutture. Ad esempio l'istruzione *if* è "ribaltabile", infatti un brano di codice del tipo:

```
if ($a == 1) { print "a vale 1!\n" }
```

può tranquillamente diventare:

```
print "a vale 1!\n" if $a == 1;
```

In questo caso si possono tranquillamente eliminare le parentesi, in quanto questa seconda sintassi non le richiede. Il "ribaltamento" funziona anche con *unless*:

```
print "a vale 1!\n" unless $a != 1;
```

Al posto di *if* e di *unless* si possono usare gli operatori logici. Il primo spezzone di codice può infatti essere scritto anche così:

```
$a == 1 and print "a vale 1!\n";
```

oppure così:

```
$a != 1 or print "a vale 1!\n";
```

Come si nota le alternative sono tante. Ogni programmatore tende ad utilizzare sempre la stessa forma per la programmazione condizionale, solitamente quella tradizionale. Questo ha il vantaggio di rendere i listati più facilmente comprensibili a chi conosce meno bene il linguaggio o a chi conosce altri linguaggi, ma al contempo rende i programmi più difficili da leggere in quanto la

sintassi tradizionale impone sempre l'impiego delle parentesi. In alcuni casi può essere inoltre utile l'operatore ternario "?:", che permette di scrivere un *if-else* in maniera sintetica al massimo. Poniamo ad esempio di avere:

```
if ($a eq 'sogno') { print "Stiamo sognando\n" }
else { print "Siamo svegli\n" }
```

Esso può essere riscritto nella seguente forma:

```
$a eq 'sogno' ? print "Stiamo sognando\n" : print "Siamo svegli\n";
```

La condizione a sinistra del punto di domanda viene valutata: se essa è vera, il codice tra il punto di domanda ed i due punti viene eseguito; se al contrario essa è falsa, vengono eseguite le istruzioni poste a destra dei due punti. Questo operatore molto potente e sintetico deriva direttamente dal C: esso ha il vantaggio di permettere di implementare semplici blocchi condizionali scrivendo pochissimo codice, e lo svantaggio di rendere il sorgente più difficile da leggere.

## Blocchi condizionali

Quando i test condizionali sono molti, utilizzare le istruzioni *if-elsif-else* risulta piuttosto scomodo. Perl, obbedendo anche in questo caso alla massima "c'è più di un modo per ottenere lo stesso risultato", mette naturalmente a disposizione varie alternative. Consideriamo ad esempio il seguente blocco condizionale, scritto utilizzando la forma tradizionale:

```
if ($a == 1) { print "1!\n" }
elsif ($a == 2) { print "2!\n" }
elsif ($a == 3) { print "3!\n" }
else { print "altro valore!\n" }
```

Esso può essere scritto, ricordando i costrutti condizionali visti poc'anzi, in maniera più chiara:

```
SWITCH: {
    print "1!\n", last SWITCH if $a == 1;
    print "2!\n", last SWITCH if $a == 2;
    print "3!\n", last SWITCH if $a == 3;
    print "altro valore!\n";
}
```

In questo esempio viene creato un blocco di istruzioni che contiene tutti i test condizionali, i quali altro non sono che *if* ribaltati, già descritti in precedenza.

Ogni riga comporta l'esecuzione di due istruzioni se la condizione risulta essere vera: la stampa a video di una stringa e l'uscita dal blocco *SWITCH*, tramite *last*. Se tale istruzione non fosse utilizzata, anche le condizioni successive verrebbero controllate e, soprattutto, l'ultima riga del blocco verrebbe eseguita. Essa infatti rappresenta la (del tutto opzionale) scelta "di default", quella che cioè viene eseguita se tutti i test condizionali precedenti falliscono: di fatto l'ultima riga corrisponde al blocco *else* di *if-elsif-else*.

È importante notare che *SWITCH* rappresenta puramente un nome per il blocco di istruzioni contenuto tra le parentesi graffe: al posto di esso si può usare qualunque altra parola. L'utilizzo di

*SWITCH* in particolare è piuttosto diffuso poiché il risultato di quanto sopra riportato corrisponde a ciò che si può ottenere con l'istruzione `switch` presente in linguaggi quali ad esempio il C.

All'interno del blocco è possibile utilizzare una qualsiasi delle sintassi condizionali viste nella primissima parte di questa lezione. La scelta dipenderà dal livello di chiarezza che si desidera dare al codice, alla comodità di utilizzo e, non ultimo, alle proprie preferenze personali.

## Strutture iterative: `for`, `while`, `until`

Utilizzando le strutture iterative è possibile ripetere l'esecuzione di un blocco di codice Perl per un certo numero di volte senza doverlo sempre riscrivere. Il numero di iterazioni è essenzialmente determinato da due fattori: esso può essere fisso (cioè predeterminato) oppure arbitrario, deciso dall'avverarsi o meno di una condizione. La struttura iterativa più nota era anche l'unica presente nei primi linguaggi di programmazione. Essa prevede l'impiego dell'istruzione *for*. Vediamone subito un esempio:

```
for ($i = 0; $i <= 3; $i++) {  
    print "2 x $i = " . (2*$i) . "\n";  
}
```

Questo scampolo di codice esegue un certo numero di volte, per la precisione 4, la riga contenuta tra le parentesi graffe. L'output del programma sarà:

```
2 x 0 = 0  
2 x 1 = 2  
2 x 2 = 4  
2 x 3 = 6
```

e cioè la stampa degli operandi e del risultato della moltiplicazione di 2 per un numero che va da 0 a 4. Questo numero è il valore della variabile *\$i*, detta variabile di controllo del ciclo; esso cambia ad ogni ripetizione del ciclo, incrementandosi sempre di una unità.

L'istruzione `for` accetta infatti, tra parentesi, tre parametri: il primo definisce il valore iniziale della variabile di controllo (0 in questo caso), il secondo rappresenta la condizione necessaria per l'esecuzione del blocco di istruzioni interno (il ciclo viene eseguito fino a che tale condizione risulta essere vera, in questo caso finché *\$i* è minore o uguale a tre), il terzo indica invece la variazione da apportare alla variabile di controllo (incremento di una unità in questo esempio). Se si volessero invece saltare tutti i numeri dispari si potrebbe scrivere:

```
for ($i = 0; $i <= 3; $i = $i+2) {  
    print "2 x $i = " . (2*$i) . "\n";  
}
```

Se l'incremento che si desidera dare a *\$i* è sempre una unità, esiste una forma più semplice di *for*, che prevede come parametri il solo numero iniziale e quello finale. Essa prevede l'utilizzo dell'operatore `..`, già studiato in relazione agli array.

```
for $i(0..3) {  
    print "2 x $i = " . (2*$i) . "\n";  
}
```

Il *while* è l'altra fondamentale struttura iterativa. Vediamo come trasformare il ciclo *for* di cui sopra in un blocco *while*:

```
$i = 0;
while ($i <= 3) {
    print "2 x $i = " . (2*$i) . "\n";
    $i++;
}
```

La variabile *\$i* va in questo caso impostata al valore iniziale fuori dalla parte iterativa. Tutto ciò che fa *while* è valutare se una condizione è vera o falsa e continuare ad eseguire il ciclo finché essa risulta essere vera. In questo caso viene utilizzata *\$i*, che però non è una variabile di controllo come nel ciclo *for*: la sua presenza è del tutto indifferente a *while*, ed infatti essa va gestita manualmente: il suo incremento in questo caso è ottenuto tramite la riga *\$i++* inserita nel blocco di istruzioni interno al ciclo iterativo. Come il significato logico di *if* era ribaltabile con *unless*, quello di *while* è a sua volta invertibile utilizzando *until*. Il codice appena scritto diventerebbe in questo caso:

```
$i = 0;
until ($i > 3) {
    print "2 x $i = " . (2*$i) . "\n";
    $i++;
}
```

In altre parole l'esecuzione di *while* termina quando la condizione è falsa, mentre l'esecuzione di *until* s'interrompe quando la condizione risulta vera. Una forma un tantino diversa di queste due istruzioni permette di far sì che il ciclo venga comunque eseguito una volta prima che la condizione venga testata:

```
$i = 0;
do {
    print "2 x $i = " . (2*$i) . "\n";
    $i++;
} while $i <= 3;
```

Senza utilizzare il *do*, se la condizione è falsa in partenza (poniamo che *\$i* sia impostato ad esempio a 10), il ciclo non viene mai eseguito: con *do* si ha invece almeno una iterazione in ogni caso.

L'esecuzione del blocco all'interno delle strutture iterative può anche essere controllata direttamente da tale codice, senza intervenire sulla condizione, grazie a *last*, *next* e *redo*. La prima istruzione, già vista per quanto riguarda *SWITCH*, causa l'immediata interruzione dell'esecuzione del ciclo, con conseguente passaggio del controllo alla prima riga posta sotto di esso. Le seconde due causano l'immediata interruzione dell'esecuzione dell'iterazione corrente e la ripresa del ciclo dall'inizio. Tra le due istruzioni c'è una sottile ma importantissima differenza: con *next* la condizione viene valutata nuovamente, con *redo* no. In un ciclo *for* questo comporta che utilizzando la prima istruzione la variabile di controllo venga incrementata, cosa che non avviene se si è invece usato *redo*. Vediamo qualche riga di esempio: un esempio di tutto questo è visibile in **Listato 1**.

```
$ok = 0; # Definita per evitare il loop infinito
```

```

for ($i = 0; $i <= 3; $i++) {

    # Salta 0
    if ($i == 0) { next; }

    # Se $i == 2 esce
    if ($i == 2) { last; }

    # Stampa
    print "2 x $i = " . (2*$i) . "\n";

    # Se $i == 1 ripete (stampa due volte)
    # a meno che $ok == 1
    if (($i == 1) && ($ok == 0)) { $ok=1; redo; }
}

```

L'output di questo programma sarà: dunque:

```

2 x 1 = 2
2 x 1 = 2

```

Qualcuno avrà a questo punto intuito che è ad esempio possibile implementare un `if` utilizzando un *while*. Ad esempio:

```

if ($a == 1) { print "a vale 1\n"; }

```

può tranquillamente diventare:

```

while ($a == 1) { print "a vale 1\n"; last; }

```

Quanto appena scritto dimostra solamente ancora una volta che ci sono molti metodi per ottenere il risultato che si cerca; si tratta solo di saper scegliere quello più adatto alla situazione. Sostituire un *if* con un *while* non è mai una buona idea: oltre ad essere inelegante, è meno efficiente.

## Analisi di array ed hash con foreach

L'ultima struttura iterativa rimasta da spiegare è *foreach*, che permette di scandire un array oppure un hash elemento per elemento. Poniamo di aver definito il seguente array:

```

@luoghi = ('Roma', 'Berlino', 'New York', 'Melbourne');

```

Come si potrebbe fare a stampare tutti gli elementi dell'array? La soluzione più semplice e più inelegante è quella di servirsi di quattro istruzioni *print*. Si potrebbe alternativamente utilizzare un ciclo *for*:

```

for $i(0..$#luoghi) {
    print "$luoghi[$i]\n";
}

```

Ricordandosi che `$#luoghi` contiene l'indice dell'ultimo elemento dell'array, si capisce subito che questo ciclo viene ripetuto tante volte quanti sono gli elementi di `@luoghi`, stampandone uno per ogni iterazione. L'utilizzo di questa struttura iterativa è una buona soluzione, ma in Perl si può anche scrivere:

```
foreach $l(@luoghi) {
    print "$l\n";
}
```

Questa struttura memorizza in *\$l* un elemento alla volta dell'array, e fa sì che le istruzioni nel blocco interno vengano eseguite finché tutta la lista è stata analizzata. Una delle comodità di *foreach* è quella rendere possibile l'analisi di una copia modificata dell'array, senza per questo alterare l'originale. Ad esempio:

```
foreach $l(sort @luoghi) {
    print "$l\n";
}
```

causerà la stampa di tutte le città ordinate alfabeticamente. In realtà *@luoghi* non viene assolutamente variato: ne viene creata ed utilizzata una copia anonima, contenente gli elementi ordinati. Lo stesso principio permette di analizzare gli hash. Consideriamo il seguente array associativo:

```
%capitali = qw (Germania Berlino
                Italia Roma
                Brasile Brasilia
                Irlanda Dublino);
```

Per stampare tutti i nomi delle nazioni e delle rispettive capitali si può scrivere:

```
foreach $k(keys %capitali) {
    print "$k: $capitali{$k}\n";
}
```

L'output sarà formato da quattro righe ciascuna contenente il nome di una nazione seguito dal nome della capitale di essa:

```
Germania: Berlino
Italia: Roma
Brasile: Brasilia
Irlanda: Dublino
```

Il programma analizza un array normale contenente le chiavi dell'hash che abbiamo creato e memorizza in *\$k* ciascuna di esse: è poi necessario richiamare l'array associativo originale per la stampa della capitale. Si può anche facilmente ottenere una lista ordinata per nazione:

```
foreach $k(sort keys %capitali) {
    print "$k: $capitali{$k}\n";
}
```

La funzione *keys()* crea un array anonimo contenente le chiavi di *%capitali*: esso viene passato come parametro a *sort()*, che inizializza un'altra lista che altro non è che la copia ordinata del primo array anonimo. Anche in *foreach* si possono tranquillamente utilizzare le istruzioni *last*, *next* e *redo*: esse infatti sono applicabili indiscriminatamente a qualsiasi ciclo iterativo.

## Conclusioni

Programmazione condizionale e strutture iterative sono, come si può facilmente intuire, uno dei

fondamentali della programmazione in qualsiasi linguaggio;, quindi, esse vanno imparate molto bene. Scrivendo un programma di media lunghezza si usano di solito decine, forse centinaia, di istruzioni condizionali e svariati cicli iterativi.

# 5. Reference, subroutine, parametri

Lo scopo di questa lezione è quello di introdurre le subroutine, cioè strutture che permettono di "spezzare" il codice in più parti completamente distinte e riutilizzabili. Verrà tuttavia prima spiegato il funzionamento delle reference, cioè particolari variabili che contengono un riferimento ad un'altra variabile o struttura dati. Esse sono utili in moltissimi casi, tra cui il passaggio di parametri alle subroutine.

## Introduzione

Per programmi di una certa complessità è quasi obbligatorio, sia per esigenze di efficienza che di chiarezza, spezzare il proprio codice in più sezioni distinte, richiamabili l'una dall'altra. Spesso è anche opportuno dividere il proprio codice in più file, cosa possibilissima e che verrà spiegata nelle lezioni successive. Subroutine e funzioni, che in Perl sono praticamente la stessa entità, permettono appunto queste divisioni. Prima di addentrarci nel loro studio è tuttavia opportuno analizzare il funzionamento delle reference, che ritorneranno utili sia per quanto riguarda le subroutine che in molti altri casi.

## Introduzione alle reference

Scopriamo brevemente cosa sono e come funzionano le reference. Come è già noto da alcune lezioni, Perl mette a disposizione più tipi di variabili. Consideriamo una variabile scalare: essa contiene un numero oppure una stringa, ad esempio:

```
$a = 'bored';
```

Essa può essere direttamente modificata, stampata e via dicendo. È inoltre possibile creare un riferimento ad essa, cioè una seconda variabile che "punti" a `$a` ma che non sia `$a`. In altri linguaggi ed in altri contesti tali riferimenti sono chiamati handle: questo nome, che significa maniglia, rende probabilmente meglio l'idea di cosa si tratti: in effetti tali variabili sono delle maniglie tramite le quali è possibile accedere alla variabile originale ed al suo valore, leggendolo oppure modificandolo. Se volessimo creare una reference `$ra` a `$a` potremmo scrivere:

```
$ra = \ $a;
```

Il backslash (`\`) restituisce una reference alla variabile posta alla sua destra, valore che poi viene memorizzato in `$ra`. Se noi a questo punto scriviamo:

```
print $ra;
```

non otteniamo, come qualcuno potrebbe immaginare, la stampa del valore originale della variabile, bensì qualcosa del tipo:

```
SCALAR(0x80d1fbc)
```

La prima parte indica il tipo di variabile a cui la reference "punta". I vari tipi possibili sono visibili in **tabella 1**:

## Simbolo Significato

SCALAR Variabile scalare

ARRAY Array

HASH Hash (array associativo)

CODE Codice Perl (ad esempio una subroutine)

GLOB Typeglob

REF La reference punta ad un'altra reference, che a sua volta poi punterà ad altro

alcuni di essi non risulteranno chiari a questo punto del corso, ma lo saranno in futuro. La seconda parte della stringa visualizzata indica invece l'indirizzo della variabile puntata: la reference in questo caso si riferisce al valore memorizzato in *0x80d1fbc*. Per conoscere il tipo di una variabile puntata da una reference è anche possibile utilizzare la funzione `ref()`, che restituisce una stringa come quelle di tabella 1, omettendo l'indirizzo. Le reference si possono assegnare e copiare esattamente come le variabili normali:

```
$ra2 = $ra;  
$ra3 = \ $a;
```

*\$ra2* e *\$ra3* sono esattamente uguali a *\$ra*, in quanto la prima è stata copiata direttamente da *\$ra* e la seconda è stata creata a partire dalla stessa variabile scalare. È importante notare che non è stato copiato il contenuto di *\$a*, del quale esiste ancora una sola copia. Cancellare o modificare una delle tre reference create non altera assolutamente il valore della variabile originale. Ma quindi come si può accedere alla variabile puntata? "Dereferenziando", come nel seguente esempio:

```
print $$ra;
```

Osserviamo bene quanto scritto, che si potrebbe anche porre nella forma:

```
print ${$ra};
```

Tra parentesi graffe è indicata la reference, il cui valore è poi passato ad un secondo dollaro (*\$*), che indica che il valore da stampare è un altro scalare. L'output è prevedibilmente:

```
bored
```

L'assegnazione di un valore alla variabile puntata avviene esattamente secondo la stessa modalità:

```
$$ra = 'dark';
```

## Reference ad array ed hash

Anche ad array ed hash è possibile creare reference, utilizzando esattamente lo stesso metodo valido per gli scalari. Le seguenti due righe creano, nell'ordine, una reference ad un array ed una ad un hash, dopo averli definiti:

```
@a = (1, 2, 3);  
%b = ('a' => 'Bianco', 'b' => 'Grigio');  
$ra = \@a;  
$rb = \%b;
```

La sintassi usata è, come si vede, esattamente quella vista in precedenza, con la differenza che accanto al backslash, nella riga che definisce la reference, va riportato il simbolo dell'array oppure quello dell'hash (rispettivamente @ e %). È anche possibile creare una reference ad un array oppure ad un hash anonimo, cosicché non sia necessario definirlo prima e costruire l'handle poi:

```
$ra = [1, 2, 3];  
$rb = {'a' => 'Bianco', 'b' => 'Grigio'};
```

La differenza sostanziale con la definizione di un array o di un hash normale è che, mentre nel primo caso si utilizzano sempre le parentesi tonde, qui vanno impiegate rispettivamente quelle quadre e quelle graffe. Utilizzando questa sintassi avremo *\$ra* che punta ad un array anonimo (non accessibile direttamente tramite *@a*) e *\$rb* che punta ad un hash anonimo. Per accedere alla struttura dati si può procedere come visto per gli scalari, ad esempio si può scrivere:

```
# Stampa il numero di elementi dell'array puntato da $ra  
print scalar @$ra;  
# Stampa il numero di chiavi dell'hash puntato da $rb  
print scalar keys %$rb;
```

L'utilizzo di *scalar*, già visto nelle precedenti lezioni, è indispensabile per far sì che venga stampato il numero di elementi: se non venisse forzato un contesto scalare verrebbe visualizzata la lista degli elementi dei due array. Per accedere agli elementi di un array oppure di un hash utilizzando una sua reference si utilizza:

```
print $$ra[1];  
print $$rb{'b'};
```

Perl mette naturalmente a disposizione un'altra sintassi, più simile a quella del C++, per dereferenziare array ed hash:

```
print $ra->[1];  
print $rb->{'b'};
```

Benché il primo metodo sia il più utilizzato (il risparmio di un carattere è determinante per il pigro programmatore Perl), senza dubbio il secondo è consigliabile in quanto più chiaro e leggibile.

## Strutture dati multidimensionali

Un utilizzo immediato delle reference ad array ed hash è quello che porta alla creazione di array multidimensionali, hash multidimensionali, array di hash e hash di array. Ciascun elemento di un array o di un hash può infatti contenere una reference: se in particolare esso contiene un handle ad un'altra struttura dati, ecco allora che si capisce come sia possibile ottenere la multidimensionalità. Nel listato 5.1 (capitolo 11), sono mostrati quattro esempi, ciascuno dei quali definisce una delle strutture dati di cui sopra. Come si nota si va a creare sempre un semplice array o hash, e non una reference ad esso: è all'interno delle parentesi tonde che vengono definite, come elementi, più reference. Accedere agli elementi è piuttosto semplice, ed è possibile farlo anche senza prima definire la struttura dati: in Perl infatti, come noto, non è necessario dichiarare le variabili, ed è quindi possibile assegnare un valore ad esse senza altre istruzioni. In questo caso, rifacendoci ai quattro esempi del riquadro, potremmo scrivere:

```
$persone[1][1] = "402, Cedars Drive";
$persone2[1]['indirizzo'] = "402, Cedars Drive";
$persone3{'USA'}[1] = "402, Cedars Drive";
$persone4{'USA'}{'indirizzo'} = "402, Cedars Drive";
```

È dunque sufficiente specificare gli indici (o le chiavi nel caso si tratti di hash) da sinistra verso destra, a partire da quello più "esterno". Qualcuno potrebbe chiedersi: ma non serve dereferenziare per accedere al valore puntato dalle reference interne alla struttura dati? In realtà sì, ma Perl, nel caso di strutture multidimensionali, permette di "dimenticarselo" e lo fa in automatico. Di fatto scrivere:

```
$persone[1][1] = "402, Cedars Drive";
```

è solo un modo più pratico per scrivere:

```
_${persone[1]}[1] = "402, Cedars Drive";
```

oppure:

```
$persone[1]->[1] = "402, Cedars Drive";
```

## Subroutine

Le subroutine sono delle strutture, messe a disposizione da tutti i linguaggi di programmazione, che, come anticipato prima, permettono di spezzare il codice in più blocchi completamente distinti, e richiamabili uno dall'altro. Tutto questo ha il vantaggio di rendere il codice più chiaro e, soprattutto, di scrivere parti riutilizzabili, cioè righe di codice che possono essere richiamate un numero indefinito di volte da altri punti del proprio programma. Supponiamo di voler scrivere del codice che stampi alcune stringhe antepoendo ad esse "Nome: ":

```
print "Nome: Markus\n";
print "Nome: Michele\n";
print "Nome: Annie\n";
```

Se noi ora volessimo cambiare "Nome: " con "Persona: " ci troveremmo a dover intervenire sul codice in tre punti. Tra le varie alternative a questo, c'è quella che prevede l'utilizzo di una subroutine:

```
sub pnome {
    my $nome = shift;
    print "Nome: $nome\n";
}

pnome("Markus");
pnome("Michele");
pnome("Annie");
```

La parola chiave *sub* è quella che definisce la subroutine, che si chiama in questo caso *pnome()*: il codice che la compone va racchiuso tra parentesi graffe. Essa è poi richiamata tre volte dal codice principale, tramite le righe *pnome()*. Tra parentesi viene passato, come *parametro*, il nome da stampare. Esso viene poi memorizzato in *\$nome* nella subroutine dalla riga:

```
my $nome = shift;
```

Ma perché viene utilizzato *shift*? I parametri passati alle subroutine vengono inseriti nell'array `@_`: visto che questo è il nome della variabile "di default" di Perl, non serve passarlo come parametro a *shift()*. Volendo avremmo potuto scrivere, in maniera più completa:

```
my $nome = shift @_;
```

Come visto in una delle precedenti puntate, *shift()* restituisce il primo (ed in questo caso unico) elemento dell'array, rimuovendolo dalla lista. I parametri passati possono essere un numero arbitrario:

```
pnome("primo", "secondo", "terzo");
```

e possono essere recuperati dall'array ad esempio con:

```
my ($p1, $p2, $p3) = @_;
```

Anche in questo caso il parametro a *split()* è opzionale, in quanto aggiunto automaticamente dall'interprete. A questo punto qualcuno si starà però chiedendo cosa sia quel *my*...

## Variabili locali

Le variabili in Perl sono, se non diversamente specificato, globali: è cioè possibile accedere al loro contenuto da qualsiasi parte del programma, sia dal corpo principale che dalle varie subroutine. Si potrebbero utilizzare sempre le variabili globali, ma spesso è consigliabile "localizzarne" l'uso, sia per ragioni di efficienza che per ragioni di praticità. La parola chiave *my* permette di definire delle variabili valide solo all'interno del blocco di istruzioni corrente, cioè racchiuso tra parentesi graffe. Alla fine dell'esecuzione di tale blocco queste variabili vengono distrutte, e la memoria precedentemente allocata per esse liberata. Un altro effetto di *my* di cui è necessario tenere conto è il fatto che essa maschera un'eventuale variabile globale con lo stesso nome. Chiariamo con un esempio:

```
sub mysub {
    my $a = 1;
    print "$a\n";
}

$a = 2;
print "$a\n";
mysub();
print "$a\n";
```

L'output di questo programma è:

```
2
1
2
```

Il valore della variabile globale *\$a* non viene quindi cambiato, poiché *my* crea una variabile con lo stesso nome, ma completamente distinta e valida solamente all'interno della subroutine. Se non avessimo utilizzato *my* ma solamente:

```
$a = 1;
```

il valore della variabile globale sarebbe effettivamente stato cambiato e l'output sarebbe di conseguenza stato:

```
2  
1  
1
```

Si può subito intuire l'estrema utilità di questa caratteristica: è possibile infatti scrivere subroutine utilizzando variabili con nomi qualsiasi, senza preoccuparsi di sceglierle in modo che non coincidano con altre già utilizzate nel programma.

## Parametri

Abbiamo visto, nella prima subroutine analizzata, come si fa a passare uno o più parametri. Consideriamo nuovamente una subroutine di questo tipo:

```
sub incrementa {  
    my $num = shift;  
    $num++;  
}
```

Questo (inutile) codice incrementa semplicemente di una unità il parametro. Se eseguiamo la chiamata come di seguito indicato:

```
$mionumero = 4;  
incrementa($mionumero);  
print $mionumero;
```

ci ritroviamo con un *\$num* che vale sempre 4, anche dopo l'incremento. La ragione di questo è abbastanza chiara: Il parametro passato viene copiato in *\$num*, che è una variabile locale della funzione: è quindi questa ad essere modificata, e non quella esterna. Anche ponendo che variabile esterna ed interna avessero avuto lo stesso nome, la sostanza non sarebbe cambiata, per le motivazioni viste poc'anzi. Avremmo potuto risolvere togliendo il *my* ed utilizzando semplicemente il seguente programma:

```
sub incrementa {  
    $mionumero++;  
}  
  
$mionumero = 4;  
incrementa();  
print $mionumero;
```

Questo, benché funzionante, è molto limitativo: se noi avessimo avuto più di una variabile da incrementare, avremmo dovuto scrivere tante subroutine quante erano tali variabili. Si intuisce subito l'inefficienza e la scarsa praticità del tutto. Perl consente tuttavia di modificare il valore dei parametri, a patto che anziché passare direttamente la variabile alla funzione ne venga passata una reference. Vediamo un esempio:

```
sub incrementa {
```

```
my $rnum = shift;
$$rnum++;
}

$mionumero = 4;
incrementa(\$mionumero);
print $mionumero;
```

La chiamata a *incrementa()* prevede in questo caso, tramite l'uso del backslash (\) la creazione "al volo" di una reference a *\$mionumero* ed il passaggio di questa alla subroutine. All'interno di quest'ultima, tale reference viene memorizzata in *\$rnum* e successivamente dereferenziata ed incrementata con *\$\$rnum++*. L'output sarà correttamente 5.

## Conclusioni

Abbiamo iniziato a capire come rendere il proprio codice più chiaro, efficiente e soprattutto riutilizzabile tramite l'impiego delle subroutine, che portano ad un tipo di programmazione più "strutturata". A questo punto inizia inoltre ad apparire chiara l'utilità delle reference: esse sono indispensabili anche per il passaggio di parametri non scalari come hash ed array, argomento che studieremo nella prossima lezione. Analizzeremo inoltre le funzioni, cioè delle subroutine che restituiscono un valore.

# 6. Funzioni e regular expression

**In questa lezione termineremo anzitutto l'analisi di subroutine e funzioni, studiando cosa rappresentano queste ultime e l'utilizzo delle reference per il passaggio di parametri non scalari. La seconda parte sarà invece un'introduzione alle regular expression, che rappresentano una delle caratteristiche più comode e potenti in assoluto del linguaggio Perl.**

## Introduzione

Nella scorsa lezione sono state illustrate le reference e le subroutine. Le prime sono risultate utili per passare dei parametri alle subroutine facendo in modo che fosse possibile modificarne il valore dall'interno della subroutine chiamata. Ciò che ancora manca da analizzare è come passare alle subroutine parametri non scalari (ad esempio array ed hash) e come ottenere da esse un valore restituito, argomenti che tratteremo tra poche righe. Inizieremo inoltre ad imparare l'utilizzo di alcune caratteristiche di Perl dedicate all'elaborazione di dati, in particolare di testi o comunque di strutture leggibili dall'uomo. Verranno infatti introdotte le *regular expression*, tramite le quali, superato lo shock dell'apprendimento iniziale, è possibile effettuare l'analisi di testi in maniera comoda e veloce.

## Parametri non scalari

Dovrebbe ormai essere chiaro come passare un parametro scalare ad una subroutine. Ma come fare a passare ad esempio un array? A prima vista può sembrare logico procedere come segue:

```
sub mysub {
    my @a = @_;
    print @a;
}

mysub (@mioarray);
```

Ragionando un momento si capisce subito che questo codice non funziona come desiderato, in quanto i parametri costituiscono un array essi stessi e quindi, ponendo che *@mioarray* contenga tre elementi, la chiamata vista sopra è di fatto equivalente a:

```
mysub ($mioarray[0], $mioarray[1], $mioarray[2]);
```

L'array passato è quindi visto come una lista di parametri anziché come uno unico. Ci vengono fortunatamente in aiuto le reference. È sufficiente infatti scrivere:

```
mysub (\@mioarray);
```

e verrà correttamente passato l'array a *mysub()*. Più precisamente ciò che la procedura si ritrova come parametro è una reference, quindi la subroutine dovrà essere modificata come segue:

```
sub mysub {
    my $aref= shift;
    my @a = @$aref;
    print @a;
```

```
}
```

Allo stesso modo è possibile passare ad una funzione un hash, oppure più di uno, o un array ed un hash, oppure ancora un array ed una lista di scalari. Ad esempio potremmo avere:

```
mysub (\@mioarray, $miastringa, \%miohash);
```

I parametri possono essere poi recuperati all'interno della subroutine con:

```
($aref, $stringa, $href) = @_;
```

A questo punto l'utilità delle reference è chiara: senza di esse non sarebbe possibile l'implementazione di alcune strutture dati quali gli array multidimensionali e sarebbe precluso un passaggio di parametri come quello visto poc'anzi. In realtà anche prima che Perl 5 introducesse queste strutture era possibile ovviare, in maniera sicuramente più scomoda, utilizzando ad esempio i `typeglob`, che ora servono solo in rari casi.

## Funzioni

In tanti casi è necessario poter ottenere un valore restituito da una subroutine. Se ad esempio dovessimo elevare un certo numero al cubo potremmo volere utilizzare una sintassi tipo:

```
$a = 3;  
$b = cubo($a);
```

Se invece lo scopo fosse quello di effettuare un controllo ortografico su una stringa potremmo voler scrivere:

```
# Torna 0 se OK, 1 se errore  
$a = 'libro';  
$b = controlla($a);
```

Nel primo caso l'obiettivo è quello di ottenere il cubo di un numero senza alterare il valore della variabile originale, mentre lo scopo del secondo listato è quello di avere un responso da una subroutine che effettua un determinato controllo. Per questo scopo esistono le *funzioni*, che in realtà altro non sono che subroutine in tutto e per tutto, con la differenza che all'interno di esse è inserita un'istruzione che permette di restituire un valore al codice chiamante. Vediamo ad esempio come potrebbe essere implementata la funzione di elevazione al cubo:

```
sub cubo {  
    my $n = shift;  
    my $c = $n*$n*$n;  
    return $c;  
}
```

L'istruzione chiave è *return*, che ha un duplice effetto: termina immediatamente l'esecuzione della subroutine e restituisce un valore (il contenuto di `$c` in questo caso) al codice chiamante. Qualsiasi riga di codice posta sotto un `return` non viene quindi eseguita, e si potrebbe infatti migliorare l'efficienza di `cubo()` in questo modo:

```
sub cubo {
```

```

my $n = shift;
if ($n == 0) {
    return;
}
my $c = $n*$n*$n;
return $c;
}

```

In questo caso il primo utilizzo di `return` è senza parametro, il che fa sì che la funzione torni il valore indefinito (*undef*) se utilizzata in contesto scalare, un array vuoto se utilizzata in contesto di lista, e nulla se utilizzata fuori contesto (cioè come una subroutine). È permesso specificare un array oppure un hash come valore restituito, sempre utilizzando le reference:

```
return \%miohash;
```

Intuitivamente, all'interno del codice chiamante la variabile non scalare può essere ottenuta con:

```

$href = myfunc();
%h = %$href;

```

## Introduzione alle regular expression

Le *regular expression* (spesso chiamate semplicemente *regex* e solo a volte tradotte in *espressioni regolari*) rappresentano un potentissimo strumento per la manipolazione di dati: stringhe, numeri oppure anche dati binari. Una di esse a prima vista può sembrare un'incomprensibile fila di simboli senza senso, mentre in realtà un minimo di studio permette di comprenderne almeno i caratteri generali, e di rendersi conto della loro potenza. Con una regular expression è possibile sintetizzare in una sola riga di codice ciò che altrimenti ne potrebbe richiedere svariate decine. Iniziamo da un'espressione di media difficoltà, limitandoci unicamente a mostrarla per il momento:

```
/^([\w\-\+\.\.]+)@([\w\-\+\.\.]+)\.([\w\-\+\.\.]+)$/
```

Questa regular expression controlla (con qualche limitazione) la validità sintattica di un indirizzo e-mail. Essa risulterà incomprensibile a molti, ma la sua visione è necessaria a far capire a cosa ci si trova solitamente davanti quando si legge del codice Perl, e ad infondere nel lettore una sensazione mista di fascino e di angoscia. Partiamo ora dalle cose semplici:

```

if ($a =~ m/Anna/) {
    # Fa qualcosa
}

```

Il codice di cui sopra altro non fa che controllare se `$a` contiene da qualche parte il nome *Anna*. Anzitutto va notato che l'operatore che lega la variabile all'espressione è un uguale seguito da una tilde (`=~`). La regular expression in questo caso è un po' più leggibile:

```
m/Anna/
```

La prima lettera indica il tipo di operazione: ci sono parecchie possibilità, che verranno analizzate in seguito. Per ora limitiamoci a *m*, che indica *matching*, cioè ricerca di una stringa all'interno di un'altra. Esso è l'operatore più utilizzato nelle regular expression, e può quindi essere omesso del

*tutto:*

`/Anna/`

*L'espressione logica ( $\$a \sim m/Anna/$ ) assume valore vero nel caso in cui Anna sia contenuta in un punto qualsiasi di  $\$a$ , e falso in tutti gli altri casi. Alcune situazioni in cui il valore assunto è vero sono:*

```
 $\$a = 'Anna';$   
 $\$a = 'Sono andato con Anna al cinema';$   
 $\$a = 'Annal1111';$   
 $\$a = '1111Anna';$ 
```

*Un caso particolare in cui l'espressione risulta falsa è:*

```
 $\$a = 'anna';$ 
```

*Come la comparazione normale infatti, anche quella tramite regular expression è case sensitive.*

## **Operatori interni all'espressione**

Esiste tutta una serie di caratteri speciali utilizzabili all'interno delle espressioni: una lista di quelli più generici è disponibile in tabella 1.

Simbolo	Descrizione
<code>\...</code>	Escape per i caratteri speciali
<code>... ...</code>	Alternanza ( <i>or</i> )
<code>(...)</code>	Gruppo
<code>[...]</code>	Classe di caratteri
<code>^</code>	Cerca ad inizio stringa
<code>\$</code>	Cerca a fine stringa
<code>.</code>	Trova un carattere qualsiasi

Ad esempio:

```
/^Anna/
```

restringe la ricerca del *pattern* (cioè la sottostringa) al solo inizio della stringa, quindi l'esito sarà positivo solo nel primo e nel terzo caso dei quattro esposti sopra. Naturalmente è possibile limitare la ricerca solo alla parte finale della stringa con:

```
/Anna$/
```

Ora la regular expression darà responso positivo solo nel quarto caso. È fondamentale notare che gli operatori `^` e `$` causano questo comportamento solo se inseriti rispettivamente all'inizio ed alla fine dell'espressione: in una diversa posizione o contesto hanno tutt'altra funzione. Possono anche essere utilizzati entrambi:

```
/^Anna$/
```

In questa specifica situazione il matching con regular expression ha lo stesso significato di una normale comparazione tramite *eq*. Un carattere fondamentale è il backslash, che ha la stessa funzione che esso ricopre all'interno dei doppi apici ("?"), e cioè è un escape per i caratteri speciali, che fa sì che essi vengano non vengano considerati per il loro significato; esso è inoltre un escape per il delimitatore dell'espressione (/ , se non diversamente specificato). Ad esempio:

```
/Anna\$/
```

cerca effettivamente il pattern *Anna\$*. L'operatore di alternanza (|) equivale ad un or, pertanto la seguente espressione:

```
/Anna|Camilla/
```

ha esito positivo se la stringa contiene il pattern Anna oppure Camilla oppure entrambi. Tramite le parentesi è possibile raggruppare parti di espressione, e quindi creare pattern più complessi:

```
/(Anna|Camilla|Michele) va al cinema/
```

Un matching con quest'espressione è positivo se la stringa in cui si effettua la ricerca contiene una delle seguenti sottostringhe:

```
Anna va al cinema  
Camilla va al cinema  
Michele va al cinema
```

L'operatore fondamentale di ricerca generica è il punto (.): esso trova un carattere qualsiasi ad eccezione del newline (salvo diversamente specificato). Ad esempio:

```
/^.nna$/
```

trova una stringa di quattro caratteri che finisce per *nna*, ed è quindi comodo per trovare sia *Anna* che *anna*. Si tenga però presente che per questo scopo il punto non è consigliabile (in quanto troverebbe anche ad esempio *5nna*), e che in seguito verranno illustrate delle migliori alternative.

## Quantificatori

Analizziamo ora i *quantificatori*, cioè gli operatori che permettono di decidere quante volte un determinato carattere/stringa deve essere presente nella stringa in cui effettuiamo la ricerca: essi sono visibili in **tabella 2**

Quantificatore Descrizione

*	Trova 0 o più volte (massimale)
+	Trova 1 o più volte (massimale)
?	Trova 1 o 0 volte (massimale)
{n}	Trova esattamente n volte
{n, }	Trova almeno n volte (massimale)
{n, m}	Trova almeno n volte ma non più di m volte (massimale)
*?	Trova 0 o più volte (minimale)
+?	Trova 1 o più volte (minimale)

?? Trova 1 o 0 volte (minimale)  
{n, }? Trova almeno n volte (minimale)  
{n, m}? Trova almeno n volte ma non più di m volte (minimale)

ed il loro utilizzo avviene secondo la seguente modalità:

```
/^(Anna){3}$/
```

In questo esempio il matching ha successo solo se la stringa contiene il pattern Anna ripetuto tre volte consecutive senza alcuna spaziatura e senza altri caratteri né ad inizio né a fine stringa (*AnnaAnnaAnna*). Gli altri operatori funzionano allo stesso modo:

```
# OK se Anna compare almeno 2 volte  
/^(Anna){2,}$/  
# OK se Anna compare dalle 2 alle 4 volte  
/^(Anna){2,4}$/  
# OK se Anna compare almeno una volta  
/^(Anna)+$/  
# OK se Anna compare 0 o più volte (sempre OK in questo caso)  
/^(Anna)*$/
```

Situazioni più complesse si possono analizzare combinando i vari operatori e quantificatori, ad esempio:

```
/^An{2}a.+cinema$/
```

Osservando bene questa espressione regolare si nota anzitutto che il matching avviene da inizio a fine stringa, (ci sono  $\wedge$  e  $\$$ ). In secondo luogo viene richiesto che dopo la *A* ci siano esattamente due *n* seguite da una *a*, a sua volta seguita da un numero arbitrario di caratteri qualsiasi (che però devono essere almeno uno) prima che la stringa sia terminata dalla parola *cinema*. Possibili matching con esito positivo sono quindi:

```
Anna va al cinema  
Anna è andata al cinema  
Anna andrà al cinema
```

Un esempio ancora più complesso può essere:

```
/^(An{2}a|Camilla|Michel{1,2}e).+and.+(cinema|concerto)$/
```

Oscena a vedersi, l'espressione ha risultato positivo, tra le altre, per le seguenti stringhe:

```
Michele andò al concerto  
Michelle è andata al concerto  
Camilla andrà al cinema
```

## Parentesi e quantificatori minimali

In un'espressione regolare le parentesi hanno una duplice funzione: quella di raggruppamento e quella di estrarre parte del pattern, memorizzandolo in una variabile. Poniamo di avere il seguente codice:

```
$a = 'Anna va al cinema';  
$a =~ /^(.+)\s(.+)\s$/;  
print "$1\n";  
print "$2\n";
```

L'output di questo programmino è rappresentato dal nome della persone (*Anna* in questo caso) e dal luogo dove va (*al cinema*). Il contenuto delle parentesi viene infatti memorizzato in variabili dal nome  $\$n$ , dove  $n$  ha base 1, andando da sinistra verso destra secondo l'ordine di apertura delle parentesi. Nel caso si abbia quindi un set di parentesi dentro l'altro, la numerazione parte comunque da quello più esterno, in quanto aperto prima. A questo punto qualcuno si starà però chiedendo a cosa servano gli operatori minimali di tabella 2. Poniamo di avere la seguente stringa:

```
$a = 'disse "ci vediamo" e lei chiese "quando?";
```

e di voler estrarre il contenuto della prima coppia di doppi apici. A occhio potrebbe andare bene la seguente espressione:

```
$a =~ /"(.+)"/;
```

In effetti tutto funzionerebbe se il set di apici fosse solo uno, ma nel nostro caso  $\$1$  contiene:

```
ci vediamo" e lei chiese "quando?
```

che non è esattamente quello che stiamo cercando. Questo avviene perché gli operatori sono "ingordi" (traduzione dell'inglese *greedy*) e quindi il  $+$  tende a prendere più caratteri possibili prima che venga analizzata la porzione successiva dell'espressione, tornando sui suoi passi solo nel caso il matching di tale porzione successiva fallisca. Ecco quindi chiaro lo scopo degli operatori minimali, o "non ingordi", che prendono invece meno caratteri possibile e si ottengono aggiungendo un punto di domanda dopo l'operatore massimale. Dunque utilizzando:

```
$a =~ /"(.+?)"/;
```

otteniamo regolarmente:

```
ci vediamo
```

## Conclusioni

Benché al primo impatto risultino assai ostiche, le regular expression sono uno strumento potentissimo per il programmatore.

# 7. Approfondimenti sulle regular expressions

**Continuiamo in questa lezione il nostro viaggio all'interno delle regular expression. Esse rappresentano una delle caratteristiche più potenti ed utilizzate del Perl e vanno quindi trattate con un certo approfondimento. Saper utilizzare le espressioni regolari vuol dire poter effettuare il parsing di qualsiasi stringa o file utilizzando pochissime righe di codice.**

## Introduzione

La scorsa lezione è stata di carattere più che altro introduttivo: come accennato verso la fine, ben superiore è la potenza delle regular expression, ed essa verrà almeno in parte spiegata in questa puntata. Anzitutto sarà mostrato l'uso delle classi di caratteri e di alcuni *metacaratteri* utili per effettuare raffinati pattern matching all'interno di una stringa. In secondo luogo saranno mostrate alcune funzioni diverse dalla pura ricerca di sottostringhe, quali la sostituzione di caratteri.

## Classi di caratteri

Poniamo di definire la seguente stringa:

```
$a = 'Ho comprato 2 rose nere';
```

e che il numero di rose possa variare indefinitamente. Quello che desideriamo è far sì che il matching abbia esito positivo solo se le rose comprate sono due oppure tre. La soluzione al problema potrebbe essere la seguente:

```
$a =~ /^Ho comprato (2|3) rose nere$/;
```

È tuttavia possibile ricorrere alla classi di caratteri, che come vedremo tra poco costituiscono una soluzione ben più completa a problemi di questo tipo:

```
$a =~ /^Ho comprato [23] rose nere$/;
```

La parte tra parentesi quadre è la classe, ed è costituita dai vari caratteri che dobbiamo considerare validi ai fini del matching, in questo caso 2 e 3. Una classe di caratteri permette anche di specificare un intervallo. Se ad esempio le rose devono essere un numero variabile da tre a otto possiamo utilizzare una qualsiasi delle seguenti tre soluzioni:

```
# Senza classe di caratteri
$a =~ /^Ho comprato (3|4|5|6|7|8) rose nere$/;
# Con classe di caratteri senza intervallo
$a =~ /^Ho comprato [345678] rose nere$/;
# Con classe di caratteri ed intervallo
$a =~ /^Ho comprato [3-8] rose nere$/;
```

Appare evidente la maggiore efficienza delle ultime due soluzioni, e la maggiore praticità dell'ultima in particolare. Gli intervalli specificabili sono di vario tipo:

```
# Cerca un qualsiasi numero in $a
$a =~ /[0-9]/;
# Cerca un qualsiasi lettera minuscola in $a
$a =~ /[a-z]/;
# Cerca un qualsiasi lettera maiuscola in $a
$a =~ /[A-Z]/;
```

In una classe si possono tranquillamente combinare più intervalli, mescolandoli anche a caratteri singoli. Se ad esempio poniamo un generico (e fin troppo semplice) codice di registrazione di un programma:

```
$a = 'REGt';
```

e vogliamo verificare che l'ultimo carattere sia compreso tra a e c, F e M oppure sia t o ancora un numero fra 3 e 7 possiamo usare:

```
$a =~ /^REG[a-cF-M3-7t]$/;
```

Ogni classe di caratteri può avere significato sia negativo che positivo. Se infatti il nostro desiderio è che l'ultimo carattere del codice non sia uno di quelli sopra citati possiamo senz'altro utilizzare:

```
$a !~ /^REG[a-cF-M3-7t]$/;
```

L'operatore `!~` ribalta il significato logico di ogni pattern matching: esso differisce da `=~` allo stesso modo in cui `!=` differisce da `=`. Potremmo comunque ometterne l'uso cambiando direttamente il significato della classe di caratteri:

```
$a =~ /^REG[^a-cF-M3-7t]$/;
```

L'accento circonflesso (`^`, chiamato informalmente anche *coppo*) corrisponde ad un not, e quindi indica alla regular expression di trovare caratteri che non siano quelli che lo seguono all'interno della classe. È molto importante distinguere i due diversi significati dell'accento circonflesso: se è posto all'inizio della regular expression esso fa sì che il matching abbia inizio con il primo carattere della stringa e mai dopo; se invece il suo contesto è rappresentato da una classe di caratteri, esso ne ribalta il significato. I quantificatori possono essere associati alle classi esattamente come lo sono ai caratteri normali:

```
$a =~ /^Ho comprato [0-9]+ rose nere$/;
```

Questa regular expression ha esito positivo se il numero di rose è un qualsiasi intero positivo maggiore o uguale a zero: la classe di caratteri è infatti ripetibile, per via del quantificatore `+`, un numero arbitrario di volte. Potremmo ulteriormente raffinare la nostra espressione così:

```
$a =~ /^Ho comprato [1-9][0-9]* rose nere$/;
```

La differenza con l'esempio precedente è che in questo caso la prima cifra (che può anche essere l'unica) deve essere un numero da uno a nove: non è quindi possibile evitare di acquistare almeno una rosa nera!

## Metacaratteri per il pattern matching

Le classi di caratteri sono a dire il vero utilizzate solo in alcuni casi, in quanto quelle più comuni possono essere sostituite con dei praticissimi metacaratteri, caratteri dinanzi ai quali è posto un escape (cioè un backslash). Quelli classici sono riportati in Tabella 1, in cui per ciascuno di essi è anche indicata la classe corrispondente.

<b>Simbolo</b>	<b>&gt;Significato</b>	<b>Classe di caratteri</b>
<code>\d</code>	Numero	<code>[0-9]</code>
<code>\D</code>	Non-numero	<code>[^0-9]</code>
<code>\s</code>	Spaziatura	<code>[\t\n\r\f]</code>
<code>\S</code>	Non-spazio	<code>[^\t\n\r\f]</code>
<code>\w</code>	Carattere alfanumerico	<code>[a-zA-Z0-9_]</code>
<code>\W</code>	Carattere non alfanumerico	<code>[^a-zA-Z0-9_]</code>

Scrivere ad esempio:

```
$a =~ /[0-9]{3}/;
```

equivale ad utilizzare:

```
$a =~ /\d{3}/;
```

I metacaratteri possono essere raggruppati a loro volta in una classe più complessa, ed è possibile inserire in quest'ultima anche caratteri normali. Se ad esempio vogliamo che la regular expression trovi solo numeri, spazi e cancelletti possiamo scrivere:

```
$a =~ /[\d\s#]/;
```

Chi ha dato un'occhiata a `\s` o `\S`, sempre in Tabella 1, avrà notato che la classe di caratteri equivalente ad essi contiene uno spazio ed altri quattro metacaratteri: lo spazio naturalmente trova se stesso, mentre gli altri trovano rispettivamente il *tab*, il *newline*, il *carriage return* ed il *form feed*. Tutti questi caratteri sono considerabili come spazi, e sono quindi stati inseriti in `\s` e `\S`: è opportuno tenere conto di questo aspetto quando si scrivono le proprie regular expression. In Tabella 2 è riportata la lista dei metacaratteri di uso più comune.

### **Simbolo Significato**

<code>\0</code>	Trova il carattere NULL (ASCII 0)
<code>\NNN</code>	Trova il carattere indicato, in notazione ottale, fino a <code>\377</code>
<code>\a</code>	Trova il carattere BEL (quello che fa suonare lo speaker)
<code>\b</code>	Trova il carattere BS (backspace)
<code>\cX</code>	Trova il carattere CTRL-X (es.: <code>\cZ</code> trova CTRL-Z)
<code>\d</code>	Trova un numero
<code>\D</code>	Trova un non-numero
<code>\e</code>	Trova il carattere ESC (escape, da non confondersi col backslash)
<code>\f</code>	Trova il carattere FF (form feed)
<code>\n</code>	Trova il carattere NL (new line, CR sui Macintosh)
<code>\r</code>	Trova il carattere CR (carriage return, NL sui Macintosh)
<code>\s</code>	Trova uno spazio (spazio, HT, NL, CR, FF)

<code>\s</code>	Trova un non-spazio
<code>\t</code>	Trova il carattere di tabulazione (HT)
<code>\w</code>	Trova un carattere alfanumerico più l'underscore ( <code>_</code> )
<code>\W</code>	Trova un carattere che non sia un alfanumerico o un underscore
<code>\x</code>	Trova il carattere indicato, in notazione esadecimale

## Validità sintattica di un indirizzo e-mail

A questo punto siamo in grado di capire bene la prima regular expression vista nella scorsa puntata, e cioè quella che verifica la validità sintattica di un indirizzo di posta elettronica. Essa è di seguito riportata:

```
/^([\w\-\+\.\ ]+)([\w\-\+\.\ ]+)\.([\w\-\+\.\ ]+)\$/
```

Si notino anzitutto l'accento circonflesso ed il dollaro, posti rispettivamente all'inizio ed alla fine dell'espressione e che richiedono che l'indirizzo e-mail sia tutto ciò che è contenuto nella stringa analizzata. Il pattern si può dividere in tre parti, ciascuna delle quali è costituita da un set di parentesi tonde contenenti:

```
([\w\-\+\.\ ]+)
```

Questo sotto-pattern è formato essenzialmente da una classe di caratteri, che va in cerca di qualcosa che sia un carattere alfanumerico (incluso l'underscore), un segno meno, un segno più oppure un punto, cioè tutti i caratteri ritenuti universalmente validi in un indirizzo e-mail, escludendo la chiocciolina. Il + aggiunto fuori dalle parentesi quadre fa sì che il matching sia positivo solo se vengono trovati uno o più caratteri di quel tipo. Analizzando l'espressione da sinistra verso destra, dopo il primo set di parentesi, viene cercata una (e non più di una) chiocciolina, indispensabile in un indirizzo di posta elettronica. I due set di parentesi a destra, separati da un punto, potrebbero a prima vista sembrare un doppione, in quanto il punto è già incluso nelle classi di caratteri definite. Se tuttavia non operassimo in questo modo otterremmo come validi sia, giustamente, indirizzi quali:

```
anna@mail.adriacom.com
anna@adriacom.com
```

che, erroneamente:

```
anna@adriacom
```

La regular expression così com'è impostata richiede quindi la presenza a destra della chiocciolina di due caratteri separati da un punto. Detto questo, il pattern in questione va bene nel 99% dei casi, ma lascia tranquillamente passare per validi indirizzi quali:

```
anna@?
.@.com
```

e via dicendo. Ricordo che l'effetto collaterale di questa espressione, dovuto alla presenza delle parentesi, è la memorizzazione delle varie parti dell'indirizzo e-mail in *\$1*, *\$2* e *\$3*.

## Modificatori

Come se classi, metacaratteri, operatori e quantificatori non bastassero a complicare bene le cose, ad aumentare la potenza del pattern matching concorrono i modificatori. Essi possono essere aggiunti dopo lo slash posto alla fine del pattern appunto per "modificare" in qualche modo il comportamento di esso. Nella scorsa puntata abbiamo visto che l'espressione:

```
/Anna/
```

ha esito positivo se la variabile analizzata contiene *Anna*, ma non se essa contiene *anna*, *aNna* oppure *AnNa*. Per far sì che il matching non sia sensibile alle lettere maiuscole e minuscole possiamo fare uso del modificatore *i*, e quindi scriviamo:

```
/Anna/i
```

Un altro utile modificatore è *m*, che permette a *^* e *\$* di trovare anche i newline (*\n*). Se la nostra espressione è quindi:

```
$a =~ /^Anna$/m
```

l'esito è positivo sia in caso *\$a = "Anna"* che *\$a = "\nAnna"* che *\$a = "Anna\n"* e via dicendo. I modificatori possono anche essere combinati semplicemente scrivendoli l'uno di seguito all'altro in un ordine qualsiasi:

```
/Anna/mi
```

Utile è anche *s*: esso causa che il carattere punto (*.*), il quale regolarmente trova tutti i caratteri ad eccezione del newline, trovi anche quest'ultimo. L'ultimo modificatore che analizziamo (ve ne sono in realtà altri di uso meno frequente, ed il loro studio è lasciato al lettore che vuole approfondire l'argomento) è *g*, che fa sì che il matching sia globale: esso non si ferma cioè alla prima occorrenza del pattern ma continua fino a fine stringa, trovandole tutte. Vediamo un esempio del suo utilizzo:

```
$a = "Ho incontrato Lara, Sara e Mara";  
@nomi = $a =~ /[LSM]ara/g;  
print @nomi;
```

Questo programma stampa tutte le occorrenze del pattern trovate (*Sara, Lara e Mara*), in quanto l'espressione, se usata in contesto di lista, torna in un array tutti i match. L'eventuale uso di parentesi all'interno del pattern fa sì che nella lista venga inserito solo il contenuto di esse e non l'intera stringa trovata. Lo stesso modificatore, in un contesto scalare, consente di effettuare un *matching progressivo*: utilizzando l'espressione una seconda volta la ricerca partirà dal punto in cui si era interrotta per via del matching precedente. Ad esempio:

```
$a = "Ho incontrato Lara, Sara e Mara";  
while ($a =~ /([LSM]ara)/g) {  
    print $1;  
}
```

scriverà sul terminale un output uguale al precedente caso (quando il contesto dell'espressione era

una lista), in quanto il *while* reitera la ricerca progressiva fino a che essa continua ad avere esito positivo.

## Sostituzioni con *s///*

L'operatore *m//*, nel quale come abbiamo visto la *m* può essere omessa e di fatto lo è quasi sempre, non è l'unico messo a disposizione da Perl per quanto riguarda le regular expression. Un altro molto utile è *s///*, che permette di effettuare sostituzioni all'interno di una stringa, secondo un pattern strutturalmente identico a quelli visti sinora. Vediamo un esempio del suo utilizzo:

```
$a =~ s/rosa/viola/;
```

Questa riga di codice farà sì che in *\$a* venga cercata la prima occorrenza di *rosa* ed essa venga sostituita con *viola*. Il modificatore *g* è disponibile anche in questo caso e fa sì che vengano sostituite tutte le occorrenze del pattern nella stringa. Sono possibili sostituzioni molto più complesse quali:

```
$a =~ "Ho incontrato Lara, Sara e Mara";  
$a =~ s/\wara/una ragazza/g;
```

che modifica *\$a* facendola diventare:

```
Ho incontrato una ragazza, una ragazza e una ragazza
```

Si possono anche sfruttare le parentesi e le variabili *\$n*, come nel seguente esempio, che poniamo applicato alla stringa contenuta originariamente in *\$a*:

```
$a =~ s/(\w)ara/ara$1/g;
```

La variabile *\$1* contiene la prima lettera del nome, che viene riutilizzata a destra e posta alla fine di ogni nome anziché all'inizio, trasformando *\$a* in una curiosa:

```
Ho incontrato araL, araS e araM
```

Alcuni degli altri modificatori consentiti sono *i*, *m* e *s*: essi hanno la stessa funzione che avevano per l'operatore *m//*. Un altro molto comodo è *e*, il quale causa che la parte destra dell'espressione sia considerata linguaggio Perl. Infatti il codice:

```
$a =~ s/(\d+)/sprintf("%09d", $1)/ge;
```

sostituisce tutti i numeri trovati in *\$a* con gli stessi formattati a nove cifre, anteponendo degli zeri se il numero dovesse essere più corto. Non abbiamo ancora visto l'utilizzo di *sprintf()*, che è una funzione che Perl ha ereditato dal C. Per rimuovere dalla stringa in esame dei caratteri senza inserirne altri al loro posto è sufficiente lasciare vuota la parte destra dell'espressione. Ad esempio:

```
$a =~ s/\r//g;
```

rimuove tutti i carriage return.

## Traslitterazioni con *tr///*

Questo operatore analizza una stringa, carattere per carattere, e rimpiazza tutti i caratteri della lista di sinistra con quelli della lista di destra. Esso può anche essere utilizzato come *y///*, sinonimo creato per gli utilizzatori del programma *sed*. Chiariamo subito con un esempio:

```
$a =~ tr/abcd/1234/;
```

Questo frammento cambia tutte le *a* in *1*, le *b* in *2* e via dicendo. Naturalmente sarebbe stato in questo caso più comodo scrivere:

```
$a =~ tr/a-d/1-4/;
```

È fondamentale notare che né la parte sinistra né la parte destra di *tr///* sono regular expression: esse accettano solo i metacaratteri quali `\n` per l'inserimento di caratteri non scrivibili da tastiera, ma nessun altro operatore o metacarattere proprio delle espressioni regolari. I modificatori possibili sono: *c*, che fa sì che vengano cercati i caratteri non presenti nella lista a sinistra anziché quelli presenti; *d*, il quale causa che eventuali caratteri trovati per i quali non c'è alcun rimpiazzo nella lista di destra vengano cancellati; *s*, che nel caso vengano trovati più caratteri consecutivi uguali li riduce ad uno solo, rimpiazzandolo con il rispettivo sostituto indicato nella lista di destra.

## Conclusioni

Con questa lezione abbiamo concluso la nostra panoramica sulle regular expression. In realtà esse sono degne di ulteriore approfondimento, in quanto possono far risparmiare molto tempo ad un programmatore, nonché rendere più compatto ed efficiente il codice. Nella lezione sono state omesse alcune funzionalità, la scoperta delle quali è lasciata al lettore, che può affidarsi alla corposa guida in linea fornita con Perl.

# 8. Gestione dei file e dei programmi esterni

**Qualsiasi sia il linguaggio in cui si programma, è indispensabile poter interagire in lettura e scrittura con i file di dati. Perl fornisce una serie di funzioni abbastanza potenti per fare ciò, come vedremo in questa lezione. Sarà inoltre dedicato uno spazio alle funzioni da utilizzarsi per eseguire altri programmi gestendone l'output.**

## Introduzione

Un programma ha quasi sempre necessità di accedere a dei dati. Se ad esempio ne scriviamo uno che converte un'immagine dal formato GIF allo JPEG, esso avrà bisogno di aprire l'immagine sorgente in lettura e poi memorizzarne da qualche parte l'equivalente convertito. Sono quindi necessarie delle funzioni che permettano la gestione dei file sui dischi locali: come ogni linguaggio di programmazione esistente, anche Perl ne mette a disposizione alcune. Si tratta come al solito di funzioni piuttosto potenti che permettono accessi in lettura, scrittura, simultanei o in pipe. Quest'ultimo modo permette di invocare un comando esterno gestendolo come un file per quanto riguarda l'invio dei dati di input ed il prelievo di quelli in output.

## Apertura di un file

La funzione fondamentale per l'apertura di un file Perl è `open()`, che presenta la seguente sintassi:

```
open HANDLE, MODO, NOFILE;
```

L'handle è ciò che permette di utilizzare il file una volta aperto, mentre il secondo parametro indica il modo di accesso: sola lettura, sola scrittura, append (cioè aggiunta di dati in coda) e via dicendo. Il terzo ed ultimo argomento rappresenta, abbastanza intuitivamente, il nome del file. Vediamo subito un esempio:

```
open IMG, "<", "cover.gif";
```

L'handle è solitamente una sequenza di caratteri maiuscoli: non è necessario che esso sia definito in precedenza, nemmeno se si tratta di un filehandle locale: anzi, in questo specifico caso esso non va definito, a meno che non si utilizzi una variabile al suo posto:

```
sub apri {  
    my $img;  
    open $img, "<", "cover.gif";  
    # Altro codice  
}
```

Questa sintassi, poco usata, al prezzo di una riga in più di codice per aprire il file, ha un vantaggio: esso viene chiuso automaticamente allorché il flusso giunge alla parentesi graffa di chiusura del blocco. Nel caso generale la chiusura deve invece essere compiuta esplicitamente servendosi della funzione `close()`:

```
close IMG;
```

Torniamo per un istante all'analisi dei parametri: l'utilizzo del filehandle per leggere e scrivere dati verrà illustrato in seguito; il modo di accesso specificato in questo caso è `<`, che indica che il file deve essere aperto in sola lettura. In **Tabella 1** sono riportati tutti i modi di accesso disponibili.

Modo	Letture	Scrittura	Solo append	Creazione	Cancellazione
<code>&lt;</code>	SI	NO	NO	NO	NO
<code>&gt;</code>	NO	SI	NO	SI	SI
<code>&gt;&gt;</code>	NO	SI	SI	SI	NO
<code>+&lt;</code>	SI	SI	NO	NO	NO
<code>+&gt;</code>	SI	SI	NO	SI	SI
<code>+&gt;&gt;</code>	SI	SI	SI	SI	NO
COMANDO	NO	SI	Non disp.	Non disp.	Non disp.
COMANDO	SI	NO	Non disp.	Non disp.	Non disp.

Le parentesi angolari di apertura e chiusura indicano, rispettivamente, sola lettura e sola scrittura. In quest'ultimo caso il file viene creato, con preventiva cancellazione nel caso esista già. Il modo di accesso `>>` apre invece un file in *append*: si tratta di una modalità di sola scrittura in cui però un eventuale file precedente non viene cancellato, ed i nuovi dati vengono scritti in coda ad esso; nel caso il file non esistesse viene creato, esattamente come accade con la modalità `>`. Vi sono poi i modi di accesso in lettura/scrittura: `+<<` (il file deve già esistere, in quanto non viene automaticamente creato o cancellato), `+>` (il file viene creato, ed eventualmente prima cancellato) e `+>>` (il file viene aperto in *append*, solo che è anche possibile leggervi i contenuti). Di fatto la prima modalità di lettura/scrittura è l'unica ampiamente utilizzata, mentre la seconda lo è solo di rado in quanto tutti i dati preesistenti vengono cancellati prima che sia effettivamente possibile leggerli. Essa è quindi adatta solo nel caso si voglia rileggere ciò che è appena stato scritto. È piuttosto diffusa l'usanza di unire in un'unica stringa la modalità di accesso ed il nome del file, ad esempio:

```
open IMG, ">cover.jpg";
```

Nel caso il modo venisse del tutto omissso, come in:

```
open IMG, "cover.gif";
```

il file viene aperto in sola lettura, cioè con modo `<`. La funzione `open()` restituisce un valore vero se il file viene aperto con successo e `undef` in caso contrario. Ciò rende possibile la gestione degli errori in vie come le seguenti:

```
# ### Tramite if ###
if (open IMG, "<cover.gif") {
    # Codice
    close IMG;
} else {
    die "Impossibile aprire il file: errore $!";
}

# ### Tramite or ###
open IMG, "
```

In quest'ultimo caso è importante ricordarsi di utilizzare l'operatore a bassa priorità `or` e non `//`. Se

si desidera servirsi invece di quest'ultimo è necessario mettere qualche parentesi in più:

```
open (IMG, "<cover.gif") || die ("Impossibile aprire il file: errore $!");
```

## Letture e scrittura

Leggere e scrivere su un file non è molto diverso da leggere e scrivere sul terminale. Per leggere una riga da tastiera, o comunque dallo standard input, utilizziamo:

```
$a = <STDIN>;
```

Nel caso di un file la procedura è uguale, in quanto viene sostituito l'handle del file a *STDIN*:

```
$a = <IMG>;
```

Se volessimo leggere tutte le righe in un colpo solo potremmo utilizzare lo stesso handle in un contesto di lista:

```
@righe = <IMG>;
```

Questi metodi di lettura sono tuttavia comodi più che altro quando si ha a che fare con file di testo. Come fare invece per leggere un determinato numero di byte senza che venga tenuto conto della suddivisione in righe del file? La funzione *read()* ci viene incontro. Di seguito è proposto un esempio del suo utilizzo:

```
read(IMG, $b, 10000);
```

Questo comando legge 10000 byte dal file corrispondente all'handle *IMG* e li memorizza in *\$b*. Per memorizzare l'intero file in *\$b* si può scrivere come segue:

```
read(IMG, $b, (stat(IMG))[7]);
```

A prima vista può sembrare magia nera; in realtà è un classico esempio di utilizzo di *stat()*, funzione che verrà descritta tra breve. Vediamo invece ora com'è possibile scrivere su un file che sia stato preventivamente aperto in scrittura. La funzione da utilizzarsi è la solita *print()*, per qualsiasi tipo di dato si desideri scrivere:

```
print IMG $b;
```

È quindi sufficiente indicare come primo parametro il filehandle per far sì che l'output venga rediretto ad esso. *print()* va bene sia nel caso di singole righe che di buffer letti con *read()*: non fatevi trarre in inganno dal nome della funzione, e quindi non utilizzate *write()*, che serve a tutt'altro.

## Funzioni utili

Perl incorpora svariate funzioni che semplificano la gestione di input ed output da file. Esse servono a spostarsi all'interno del file aperto e ad ottenere varie informazioni su di esso. Anzitutto *tell()* restituisce la posizione corrente, cioè il byte al quale il puntatore è attualmente collocato: se

iniziassimo a scrivere o a leggere, l'operazione avrebbe inizio da tale posizione. L'utilizzo è molto semplice:

```
$pos = tell IMG;
```

In *\$pos* si trova la posizione, che assume valore zero se il puntatore è collocato all'inizio del file. È naturalmente possibile anche modificare tale posizione tramite *seek()*, ad esempio:

```
seek IMG, 400, 0;
```

Il primo parametro è il classico handle, il secondo è l'*offset* (cioè la lunghezza dello spostamento in byte), mentre il terzo argomento è denominato *whence* ed indica l'origine dello spostamento. Questo può assumere tre valori: 0 (lo spostamento, in questo caso "lungo" 400 byte, avviene a partire dall'inizio del file), 1 (lo spostamento avviene a partire dalla posizione corrente), oppure 2 (i 400 byte vengono calcolati a partire dalla fine del file). Il potersi posizionare all'interno di un file, benché non di uso comunissimo, è fondamentale in quanto consente di leggere in maniera non sequenziale diverse parti del file, nonché di sovrascrivere i dati presenti nei vari punti di esso senza dover per forza leggerne l'intero contenuto, effettuare le modifiche, e scrivere un nuovo file. Un'altra funzione è *eof()*, che torna un valore vero se ci si trova alla fine del file. Essa è ad esempio utile se si effettuano continue letture e si vuole sapere quando smettere poiché non ci sono più dati da leggere. Ad esempio:

```
while(! eof (IMG)) {  
    # Istruzioni read, ?  
}
```

Una delle funzioni di uso in assoluto più comune Perl allorché diventi necessario gestire file esterni è *stat()*, già vista brevemente in precedenza con la promessa di spiegarla in seguito. Eccoci dunque a scoprirne il funzionamento. Il numero di informazioni fornito da *stat()* è notevole, tanto che viene tornato un array di ben tredici elementi. La sintassi è, nel caso del nostro file *IMG*:

```
@a = stat(IMG);
```

Siccome *stat()*, vista la natura di alcune delle informazioni che restituisce, è ampiamente utilizzata anche per file non aperti, essa accetta anche un nome di file anziché un handle come unico parametro:

```
@a = stat("anna.txt");
```

Ogni elemento dell'array contiene un'informazione sul file. Ad esempio l'elemento di indice sette ne indica la dimensione. Per ricavare dalla funzione solo questo dato si può scrivere, come abbiamo appunto fatto in precedenza sempre in questa lezione:

```
$size = (stat(IMG))[7];
```

La lista di tutti gli elementi dell'array ed il loro significato è riportata in **Tabella 2**.

## Indice Contenuto

0      Numero di device del filesystem

- 1 Numero di inode
- 2 Tipo e permessi del file
- 3 Numero di link (non simbolici) al file
- 4 UID numerico del proprietario del file
- 5 GID numerico del gruppo a cui appartiene il file
- 6 Identificativo del device (applicabile solo ai file speciali)
- 7 Dimensione in byte del file
- 8 Timestamp contenente data ed ora dell'ultimo accesso al file
- 9 Timestamp contenente data ed ora dell'ultima modifica al file
- 10 Timestamp contenente data ed ora dell'ultimo cambiamento all'inode
- 11 Dimensione del blocco preferibile per l'input/output
- 12 Numero di blocchi allocati per il file

Alcuni di essi probabilmente saranno incomprensibili per la maggior parte dei lettori, ma non è scopo di questo corso spiegarli. Un buon libro sui sistemi operativi Unix svelerà ogni mistero. Inoltre, per chi usa Windows, molti di essi non sono utilizzabili in quanto propri di filesystem Unix. Per quanto riguarda la dimensione di un file nello specifico è più conveniente in realtà utilizzare un operatore unario di testing, `-s`:

```
$size = -s "anna.txt";
```

Questi operatori permettono di accedere direttamente ad un'informazione, solitamente booleana (che può cioè assumere esclusivamente valore vero oppure falso) su un file. Una lista di quelli di uso più comune è disponibile in **Tabella 3**.

### Operatore Descrizione

<code>-e</code>	Vero se il file esiste
<code>-r</code>	Vero se il file è accessibile in lettura
<code>-w</code>	Vero se il file è accessibile in scrittura
<code>-d</code>	Vero se il file è una directory
<code>-f</code>	Vero se il file non è una directory o un file speciale
<code>-B</code>	Vero se il file è un file binario
<code>-T</code>	Vero se il file contiene testo
<code>-M</code>	Tempo passato dall'ultima modifica al file (timestamp)
<code>-A</code>	Tempo passato dall'ultimo accesso al file (timestamp)
<code>-s</code>	Il file non è vuoto (ha dimensione maggiore a 0 byte)
<code>-z</code>	Il file è vuoto (ha dimensione di 0 byte)

### Pipe

Le *pipe* (da pronunciarsi all'inglese, quindi come ad esempio *line*) sono, come da traduzione letterale dei *tubi* che incanalano l'output di un programma verso un altro, facendolo diventare l'input di quest'ultimo. Chi usa un sistema operativo Unix-like sa bene che ad esempio il seguente comando dato da shell:

```
cat anna.txt | lpr
```

fa sì che il file *anna.txt*, anziché essere visualizzato in console, venga passato come input ad un altro comando, *lpr*, che si preoccupa a sua volta di inviarlo alla stampante predefinita. La barra verticale è il carattere che, in una shell Unix, permette di ottenere questo comportamento. In Perl il simbolo da usarsi è esattamente lo stesso, e va indicato come modo ad *open()*. Poniamo di voler scrivere un programmino che abbia la stessa funzione della coppia di comandi appena vista. Potremmo porre:

```
open TESTO, "<anna.txt";
open STAMPA, "|lpr";
while ($riga = <TESTO>) {
    print STAMPA $riga;
}
close TESTO;
close STAMPA;
```

Questo codice apre in lettura *anna.txt* ed in pipe di input il comando *lpr*: questo è ottenuto per mezzo della barra verticale posta prima del nome del file, esattamente come viene fatto nel caso di qualsiasi altro carattere che specifica il modo, come può essere la parentesi angolare. Un normale ciclo *while* legge tutte le righe del file *TESTO* e le scrive nel file *STAMPA*. In realtà l'interprete Perl non va, ovviamente, a sovrascrivere il comando *lpr* con il contenuto di *anna.txt*, ma lo esegue passando tali contenuti al suo standard input. Le ultime due righe si occupano semplicemente di chiudere i file precedentemente aperti. Perl permette di aprire anche delle pipe in input anziché in output semplicemente ponendo la barra verticale alla fine del nome del file:

```
open ETC, "ls -l /etc|";
while ($file = <ETC>) {
    print $file;
}
close ETC;
```

In questo specifico caso viene eseguito il comando *ls -l /etc*, il quale di norma visualizza il contenuto della directory specificata, mostrando un file per ogni riga. L'output è accessibile come un normale file aperto in input.

## Comandi esterni

Oltre alle pipe Perl mette a disposizione anche un paio di metodi diretti per invocare i comandi esterni: essi sono la funzione *system()* e l'operatore apostrofo inverso. Partiamo dal secondo metodo, che è quello di uso più comune, e vediamone subito un esempio:

```
$testo = `cat anna.txt`;
```

Questo operatore provoca l'esecuzione del comando specificato all'interno degli apici inversi e la memorizzazione del suo output in *\$testo*. Analogamente è possibile servirsi della *quoted syntax*:

```
$testo = qx/cat anna.txt/;
```

A volte al posto dell'output può interessare catturare l'*exit status* di un programma. In questo caso ci viene incontro la funzione *system()*:

```
$status = system("mkdir", "mieitesti");
```

Comando e parametri vanno passati non in una stringa unica ma come una lista, quindi devono essere separati. In questo caso il programma tornerà vero se la creazione della directory è avvenuta correttamente, e falso in caso contrario. Come piccola nota, la creazione di una directory in Perl non richiede l'utilizzo di un comando esterno, in quanto è presente una funzione *mkdir()* di cui è raccomandabile l'uso. Esiste inoltre una funzione *exec()* che ha la stessa sintassi di *system()*: quest'ultima tuttavia causa la terminazione dell'esecuzione del programma corrente ed il passaggio del controllo al comando invocato, e quindi non torna mai.

## **Conclusioni**

Abbiamo visto come accedere ai file ed eseguire comandi esterni. Entrambe le operazioni sono di fondamentale importanza allorché si voglia che il proprio script sia in grado di trattare dei dati presenti su disco oppure di scrivere da qualche parte o inviare ad altro programma il proprio output. Come si nota, in Perl la gestione dei file, pur garantendo una notevole potenza e flessibilità, è alquanto semplice, anche se sono disponibili funzioni come *sysread()* e *syswrite()* che garantiscono maggior controllo ma allo stesso tempo richiedono maggiore attenzione nell'uso. Una regola generale di Perl è infatti quella di rendere facili le operazioni facili, senza rendere impossibili quelle difficili.

# 9. Le funzioni principali di Perl

**Questa lezione vuole fornire una carrellata delle più utili funzioni messe a disposizione da Perl. Il linguaggio ne offre moltissime, adatte agli usi più disparati. Qui verranno tuttavia presentate quelle più comuni ed altre molto potenti che possono tornare utili in alcuni casi specifici. Sarà dedicata maggiore attenzione a quelle relative alla gestione delle stringhe, alla matematica ed alle date.**

## Introduzione

Una delle caratteristiche salienti di Perl è il numero di funzioni disponibili senza che sia necessario ricorrere a moduli. Data una qualunque operazione da fare, difficilmente sarà necessario scrivere molto codice: nella maggior parte dei casi poche righe saranno sufficienti. Questo rende il linguaggio diverso da alcuni suoi colleghi più a basso livello, ma da esso eredita comunque molto, quali il C. Se infatti dobbiamo ordinare un array, in C sarà necessario studiarsi l'algoritmo (cosa che comunque non fa mai male, visto che pochi programmatori sembrano sapere come si effettua un mergesort oppure un *quicksort*) ed implementare una nostra funzione, o al più ricorrere a qualche libreria esterna. Perl mette invece a disposizione la funzione `sort()`, che automaticamente effettua un quicksort su una lista secondo i parametri passati. Vi sono molti altri esempi in cui Perl offre delle "scorciatoie", ed in questa lezione ne vedremo alcuni. Il fatto che una funzione sia più o meno utile naturalmente dipende dalle esigenze del singolo programmatore, che è quindi invitato a dare una rapida occhiata a tutte quelle che il linguaggio include.

## Gestione di stringhe

Una funzione di uso molto comune nel *mondo delle stringhe* è `chop()`. Il suo scopo è semplicemente quello di tagliare l'ultimo carattere della stringa passata come argomento, e solitamente viene utilizzata per rimuovere il newline (`\n`) posto al termine. Se ad esempio leggiamo, come visto nella scorsa lezione, un file riga per riga, ci troviamo ad avere una serie di stringhe terminate da newline, carattere di cui potremmo volerci sbarazzare. Scriveremmo quindi:

```
$c = chop($a);
```

Come si nota, la funzione ha anche un valore restituito, che è il carattere tagliato. È fondamentale notare che `chop()` rimuove l'ultimo carattere qualunque esso sia, quindi se per caso manca il newline, verrà tagliato l'ultimo carattere valido. C'è tuttavia l'alternativa intelligente: `chomp()`. Questa rimuove l'ultimo carattere solo se è un newline, altrimenti non ha effetto; inoltre, se i newline sono più di uno, li rimuove tutti e restituisce il numero di caratteri tagliati. La coppia `join()` e `split()` permette, abbastanza prevedibilmente, rispettivamente di unire o separare stringhe basandosi su un'espressione. Vediamo subito un esempio:

```
@a = ('Michele', 'Andrea', 'Anna', 'Luca');  
$tutti = join '_', @a;
```

`$tutti` contiene `Michele_Andrea_Anna_Luca`, cioè tutti gli elementi della lista uniti tra loro e separati da un underscore. Esattamente il contrario fa `split()`: ponendo di avere a disposizione la stringa `$tutti` appena creata, possiamo ottenere di nuovo le singole parti con:

```
@a = split /_/, $tutti;
```

Come si vede, in questo caso il primo parametro, anziché una semplice stringa come in *join()*, è un pattern: questo rendere possibile effettuare separazioni sulla base di regular expression, e quindi anche ad esempio considerando più di un carattere come separatore. Infatti:

```
split /[_;]*/, $tutti;
```

considera separatori sia l'underscore che la virgola che il punto e virgola. Simili a *join()* e *split()*, ma solo per il fatto che uniscono e separano dati, sono *pack()* ed *unpack()*. Queste sono funzioni di straordinaria potenza che, tra le altre cose, permettono di unire una lista in una stringa di lunghezza fissa: non vi sono cioè dei separatori ma un numero di caratteri fisso che determina quando finisce una parte ed inizia l'altra. Vediamo un esempio che utilizza lo stesso array *@a* di cui prima:

```
$string = pack ("A60A20A15A50", @a);
```

Il risultato è una stringa di ben 145 caratteri contenente i quattro elementi dell'array in uno schema fisso: il primo occupa 60 caratteri, il secondo 20, il terzo 15 ed il quarto 50. Visto che sono tutti più corti rispetto alla lunghezza dichiarata, il carattere A fa sì che a destra vengano inseriti degli spazi. Dovesse uno degli elementi superare la lunghezza dichiarata, la parte di esso che non sta verrebbe troncata. Tramite *unpack()* si fa l'esatto contrario:

```
@a = unpack ("A60A20A15A50", $string);
```

La funzione più importante di *pack()* ed *unpack()* non è tuttavia questa appena vista, ma quella di permettere l'accesso a file dal contenuto binario (ad esempio numeri memorizzati come tali, e non come stringhe). Chiunque fosse interessato ad accedere a questo tipo di file, troverà queste funzioni indispensabili. A volte può essere necessario creare un output un po' più "curato" di quanto *print()* permette. Allo scopo sono disponibili la funzione *write()* ed i formati, che però qui non analizziamo, ma ci limitiamo a citare in modo che il lettore interessato sappia della loro esistenza. Essi permettono di creare un output completamente formattato, gestibile in ogni suo aspetto. Vediamo invece funzioni di uso più generale, ed in particolare *printf()* e *sprintf()*. Chi conosce almeno i rudimenti della programmazione in C le ha sicuramente già incontrate, in quanto fondamentali per gestire l'input/output in tale linguaggio. Probabilmente anche chi programma in C++ le ha incontrare almeno qualche volta, benché in quest'ultimo linguaggio esse siano state di fatto sostituite dallo stream cout. *printf()* crea e stampa una stringa formattata dati i parametri di formattazione e la lista di cosa va stampato. Se ad esempio vogliamo visualizzare un numero accertandoci che sia arrotondato alla seconda cifra decimale, scriviamo:

```
printf "Il numero è: %.2f", $num;
```

Se *\$num* è *2.432874*, esso verrà visualizzato come *2.43*; di contro, se è semplicemente *2*, verrà visualizzato come *2.00*. Così come i numeri in virgola mobile, è possibile formattare qualsiasi altro tipo di dato, secondo i formati riportati in **Tabella 1**

### Carattere Descrizione

%%           Segno di percentuale

%c	Carattere corrispondente al numero indicato
%s	Stringa
%d	Intero decimale con segno
%u	Intero decimale senza segno
%o	Intero ottale senza segno
%x	Intero esadecimale senza segno
%b	Intero binario senza segno
%e	Numero in virgola mobile, notazione scientifica
%f	Numero in virgola mobile, notazione decimale fissa
%g	Numero in virgola mobile, notazione scientifica o decimale fissa
%X	Come %x, ma con lettere maiuscole
%E	Come %e, ma con la E maiuscola
%G	Come %g, ma con la E maiuscola (se presente)
%p	Puntatore (indirizzo in esadecimale)
%n	Memorizza il numero di caratteri stampati finora nella variabile passata come argomento

ed i flag visibili in **Tabella 2**.

Flag	Descrizione
spazio	Mette uno spazio prima di un numero positivo
+	Mette un + prima di un numero positivo
-	Giustifica il campo a sinistra
0	Usa zeri anziché spazi per giustificare a destra
#	Mette uno 0 prima di un ottale diverso da zero, o uno 0x prima di un esadecimale diverso da zero
num	Lunghezza minima del campo
.num	Per gli interi: lunghezza minima. Per le stringhe: lunghezza massima. Per i decimali: cifre dopo la virgola
l	Considera l'intero un long o unsigned long come in C
h	Considera l'intero un short o unsigned short come in C (senza flag considera int o unsigned int)

In questo caso il formato è *f*, ed è stato utilizzato il flag *.numero* per stabilire quante cifre dopo la virgola andassero stampate. *sprintf()* è identica a *printf()*, solo che anziché stampare la stringa creata, la restituisce:

```
$a = sprintf "Il numero è: %.2f", $num;
```

Il tipico "re dell'officina" per quanto riguarda la gestione delle stringhe è *substr()*. Questa funzione rende possibile tutta una serie di operazioni, le quali sostanzialmente permettono, come il nome della funzione lascia intuire, di gestire le parti, i singoli caratteri, di una stringa. Vediamo brevemente alcuni esempi del suo utilizzo:

```
# Taglia l'ultimo carattere (come chop())
substr($a, -1, 1) = '';
# Restituisce il terzo e quarto carattere
```

```
$b = substr($a, 2, 2);  
# Sostituisce il primo carattere con "casa"  
substr ($a, 0, 1) = "casa";
```

Come si vede gli impieghi possibili sono molteplici, e la funzione si occupa automaticamente di ridimensionare la stringa quando necessario. I parametri da passare a *substr()* sono il nome della variabile contenente la stringa, l'*offset* al quale si desidera iniziare a leggere o scrivere ed il numero di caratteri da prendere in considerazione. Un *offset* negativo indica di partire dalla fine della stringa, mentre un numero di caratteri negativo indica di considerare l'intera stringa a partire dall'*offset* lasciando però da parte quel numero di caratteri alla fine della stringa.

L'ultima e semplicissima funzione che analizziamo per quanto riguarda la gestione delle stringhe è *length()*, che restituisce il numero di caratteri di cui una stringa è composta:

```
$n = length $a;
```

## Matematica

Le funzioni matematiche messe a disposizione da Perl senza che sia necessario ricorrere a strumenti esterni sono quelle classiche, reperibili più o meno in tutti i linguaggi. Iniziamo discutendo il problema degli arrotondamenti. Esiste la funzione *int()*, che però non fa quello che molti credono: essa non arrotonda, ma semplicemente tronca la parte decimale di un numero. Per arrotondare è necessario utilizzare *sprintf()*. Ad esempio:

```
$inum = sprintf "%.0f", $num;
```

calza a pennello. Il carattere di formato *f* si occupa di arrotondare il numero decimale all'intero, approssimazione che avverrà per eccesso o per difetto a seconda del caso. Allo stesso modo è consentito arrotondare fino ad una cifra decimale a piacere, ad esempio la quarta, con:

```
$inum = sprintf "%.4f", $num;
```

Anche in questo caso è *sprintf()* a stabilire quando arrotondare per eccesso e quando per difetto. Le funzioni trigonometriche sono *sin()* e *cos()*, che restituiscono seno e coseno: tramite esse si può ad esempio ricavare, ricordando l'apposita e semplice relazione, la tangente. Esiste inoltre *atan2()*, che calcola l'arcotangente. Non c'è una funzione che restituisca  $\Pi$ , però una sua buona approssimazione la si ottiene con:

```
$pi = atan2(1,1) * 4;
```

Un'altra esigenza comune è quella di creare dei numeri casuali (*random*). La funzione *rand()* permette ciò, ad esempio:

```
$n = rand 5;
```

torna un numero in virgola mobile tra 0 e 5 (estremo inferiore incluso, estremo superiore escluso). Se non viene passato alcun parametro il numero tornato varia tra 0 ed 1, sempre escludendo l'estremo superiore. Per creare un numero casuale intero ad esempio incluso tra 1 e 20 si può quindi scrivere:

```
$n = int(rand 20) + 1;
```

Questa funzione richiama quelle di sistema per la creazione di numeri casuali, e quindi la casualità dipende da come queste ultime sono implementate. Per i casi generali essa è tuttavia sufficientemente "casuale". Una nota: in versioni di Perl precedenti alla 5.004 il *seed* dei numeri random non veniva inizializzato automaticamente. Se il proprio interprete è piuttosto vecchio, sarà necessario chiamare *srand()* all'inizio del proprio programma, *una volta sola*.

I moduli appartenenti al settore *Math*, ad esempio *Math::Trig* e *Math::Complex* permettono di effettuare qualsiasi tipo di calcolo semplicemente chiamando l'apposita funzione, e rendendo quindi semplice calcolare ad esempio seno iperbolico, secante e via dicendo, nonché effettuare conversioni e lavorare con i numeri complessi e gli interi molto grandi. Vi sono inoltre librerie che permettono di creare numeri veramente casuali (o almeno un po' più casuali). Ancora non si è tuttavia visto come utilizzare un modulo esterno, cosa che sarà argomento della prossima lezione.

## Data ed ora

È di fondamentale importanza poter gestire la data e l'ora, rendendone possibile la visualizzazione in vari formati. In Perl vi sono alcune funzioni che permettono di fare tutto ciò in maniera semplice e personalizzabile: esse sono *time()*, *gmtime()* e *localtime()*. La prima restituisce una *Unix-style timestamp* contenente il numero di secondi trascorsi dall'*Epoch* (1 Gennaio 1970, 00:00), valore dal quale è possibile ottenere tutto il resto. Infatti *localtime()* accetta come parametro uno di questi interi, chiamando automaticamente *time()* nel caso nessun parametro sia passato, e restituisce un array:

```
@ltime = localtime($ora);
```

L'array è composto come si vede in **Tabella 3**.

### Flag Descrizione

- |   |  |
|---|--|
| 0 | Numero di secondi (0-60)   |
| 1 | Numero di minuti (0-59)  |
| 2 | Numero di ore (0-23)   |
| 3 | Giorno del mese (1-31)   |
| 4 | Mese dell'anno (0-11)  |
| 5 | Anno (a partire dal 1900, quindi il valore di reale interesse è 1900+Anno) |
| 6 | Giorno della settimana (0-6, 0 è Domenica)                                 |
| 7 | Giorno dell'anno (1-366)   |
| 8 | Ora legale (true se è attiva)  |

È dunque possibile estrarre qualsiasi informazione in maniera molto semplice, e formattare data ed ora a piacere con dei semplici *print()* o *printf()*. Vediamo qualche esempio:

```
# Estrae il minuto attuale
$min = (localtime)[1];
# Estrae il giorno della settimana a tre lettere
$oggi = (Dom,Lun,Mar,Mer,Gio,Ven,Sab)[(localtime)[6]];
# Estrae l'anno
$anno = 1900+(localtime)[5];
```

*gmtime()* è uguale a *localtime()*, con l'unica differenza che, mentre la seconda torna l'ora locale, la

prima la restituisce in Universal Time (ex GMT). L'utilizzo della Unix-style timestamp può sembrare a prima vista una scelta bizzarra ed in prima battuta si tende magari ad evitarlo preferendo sempre `localtime()` o `gmtime()`. In realtà la timestamp è piuttosto comoda, in quanto permette di effettuare calcoli sull'ora utilizzando l'aritmetica tradizionale. Ad esempio per verificare che una variabile contenga una data non anteriore a 24 ore fa possiamo scrivere, tenendo conto che in un giorno ci sono 86400 secondi:

```
if ($data >= (time()-86400) {  
    # Codice  
}
```

Per memorizzare in una variabile la stessa ora di oggi, ma avanti di sette giorni è invece possibile utilizzare:

```
$futuro = time() + 604800;
```

Anche qui le possibilità, ricorrendo ai moduli, sono notevolmente superiori. In particolare è possibile costruire una timestamp a partire dai singoli valori.

## Conclusioni

In questa lezione si sono analizzate le funzioni principali offerte da Perl per quanto riguarda la gestione di stringhe, numeri e date. Per quanto riguarda settori diversi, ve ne sono naturalmente molte altre: una parte di esse verrà illustrata nella prossima lezione, le rimanenti sono lasciate allo studio individuale. Inoltre, come già accennato un paio di volte in precedenza, utilizzando dei moduli esterni (molti dei quali sono peraltro già inclusi nella distribuzione standard dell'interprete perl) si possono compiere operazioni ancora diverse, oppure le stesse in maniera più efficiente.

# 10. Altre funzioni e moduli

**Continuiamo la carrellata di funzioni analizzandone ancora qualcuna tra quelle più spesso utilizzate dai programmatori. La seconda parte della lezione è invece dedicata ad un breve introduzione ai moduli. In essa si cercherà più che altro di indicare al lettore come servirsi delle migliaia di librerie di cui la comunità Perl dispone.**

## Introduzione

Ricominciamo da dove ci eravamo fermati la scorsa lezione. Si erano viste alcune funzioni relative alla manipolazione di stringhe, numeri e date. Al di fuori di queste tre principali categorie, le funzioni offerte da Perl sono molteplici, e qui vedremo solo alcune delle più utili, peraltro in maniera molto sintetica. Rimandiamo il lettore a dare una rapida occhiata alla lista di esse reperibile nella documentazione in linea oppure in una qualsiasi guida di riferimento al linguaggio.

## Warn, exit, die!

Non si tratta dell'ultimo urlo minaccioso di qualche cantante rap-metal, bensì di una triade di funzioni che permettono di controllare l'esecuzione del proprio programma. Iniziamo dalla tenebrosa *die()*: essa generalmente causa l'immediata terminazione del programma in esecuzione; esistono tuttavia altri casi, che qui non vedremo, in cui essa può avere altro significato. Un tipico esempio del suo utilizzo è:

```
open (NOMI, "<nomi.txt") or die "Errore!";
```

*die()* stampa sullo *standard error* (*STDERR*, buffer di output dedicato agli errori, che solitamente è la console stessa) tutti i parametri passati, in questo caso la sola stringa *Errore!*, e causa la fine dell'esecuzione del programma ed il ritorno del controllo al processo chiamante. Essa va quindi usata per gestire gli errori dopo i quali è impossibile continuare. E se invece l'errore non è poi così grave ed è desiderabile solo visualizzare un messaggio sullo standard error per poi continuare regolarmente? In questo caso è possibile servirsi di *warn()*: essa è uguale a *die()* nel senso che stampa su *STDERR* tutti i suoi parametri, ma non causa alcuna terminazione del programma. Per garantire un'uscita "pulita" dal programma, senza che venga stampato alcun messaggio, va invece utilizzata *exit()*. Essa accetta anche un parametro numerico che indica lo stato di uscita (livello di errore): se questo è omissso, lo stato di uscita è 0 (nessun errore).

## Gestione di file e directory

Per semplificare e rendere più rapida la gestione dei file Perl mette a disposizione alcune funzioni di base. *mkdir()* ad esempio crea una directory:

```
mkdir "/u/mp3/SkidRow";  
mkdir "/u/mp3/SkidRow", 0700;
```

Come si vede, è possibile passare anche un secondo parametro, che indica i permessi del file. Questi sono validi solo in ambiente Unix, e chi utilizza uno di questi sistemi operativi sa bene cosa sono. Chi invece usa Windows oppure altri sistemi operativi dove i permessi non sono supportati

non ha bisogno di preoccuparsene. Una directory può naturalmente anche essere rimossa, purché vuota, con:

```
rmdir "/u/mp3/SkidRow";
```

Per leggere i contenuti di una directory vanno invece usate ben tre funzioni, in quanto il meccanismo è simile a quello relativo alla lettura dei file:

```
opendir (MPEG, "/u/mp3");  
@files = readdir (MPEG);  
closedir (MPEG);
```

La prima riga associa una directory ad un handle e la apre, la seconda ne legge il contenuto in un array ponendo un nome di file per ogni elemento, ed infine la terza chiude la directory e distrugge l'handle. Dalle directory spostiamoci ora ai file, e vediamo come rinominarli e cancellarli. La prima operazione si effettua con *rename()*:

```
rename "November_Rein.mp3", "November_Rain.mp3";
```

Il primo parametro è il nome attuale, mentre il secondo è quello che si vuole assegnare. Rimuovere un file è altrettanto facile:

```
unlink "Wasted_Time.mp3";
```

È anche possibile passare ad *unlink()* una lista di file:

```
unlink "Wasted_Time.mp3", "Eileen.mp3";
```

Si possono persino utilizzare le *wildcard* con:

```
unlink glob("*.mp3");
```

La funzione *glob()* infatti espande l'argomento passato ricavandole una lista di file, che poi viene passata come parametro ad *unlink()*.

## Grep e map

Esistono due importanti funzioni che permettono di manipolare degli array: la prima di esse è *grep()* ed essa restituisce una lista contenente solo gli elementi di quella originale che rispondono ad un determinato pattern; la seconda, *map()*, esegue invece un'istruzione o una funzione per ogni elemento di un array. Partiamo da *grep()*, che prende il nome da un programma Unix, il *GNU Regular Expression Parser*, che fa più o meno la stessa cosa. Ricordando le regular expression viste qualche lezione fa, vediamo un esempio:

```
@nomi = 'Alice', 'anna', 'Camilla', 'carlotta';  
@a =grep /^[aA]/, @nomi;
```

In *\$a* sono contenuti solo i primi due elementi di *@nomi*, e cioè quelli che iniziano per *a* oppure per *A* e per i quali il matching ha quindi esito positivo. Questa funzione è utilissima ad esempio per isolare i file a cui si è interessati a partire da una lista ottenuta con *readdir()*. Per quanto riguarda

*map()*, si tratta di una delle funzioni più potenti in assoluto di Perl: a partire da un'espressione o da un blocco di istruzioni, essa processa tutti gli elementi di una lista. Ad esempio:

```
@a = (2, 4, -4, 5, 0, -1);
@b = map { $_+1 } @a;
```

crea un array *@b* contenente tutti gli elementi di *@a* incrementati di una unità. Tra le parentesi graffe si può anche collocare un complesso blocco di istruzioni, ricordandosi che ogni elemento dell'array viene memorizzato, quando viene processato, in *\$\_*. Questa variabile, che avevamo già visto in precedenza, è particolare nel senso che molte funzioni la considerano come parametro di default se non ne vengono passati altri. Non è detto che l'array di destinazione debba avere lo stesso numero di elementi di quello di partenza, infatti:

```
@parole = map { split ' ' } @righe;
```

divide in parole ogni stringa contenuta in *@righe* e memorizza tutte queste parole separate in *@parole*. La funzione *map()* può anche essere utilizzata per svolgere il lavoro di *grep()*, infatti:

```
@a = grep /^[aA]/, @nomi;
```

è equivalente a:

```
@a = map { /^[aA]/ ? $_ : () } @nomi;
```

Per comprenderne il funzionamento è necessario ripensare alla struttura condizionale vista nella lontana quarta lezione: viene valutata l'espressione a sinistra del punto di domanda e, se il risultato logico è *vero*, viene tornato l'elemento stesso dell'array (*\$\_*), altrimenti una lista vuota.

## Utilizzo di moduli esterni

I moduli sono delle librerie tramite le quali vengono estese le possibilità offerte da Perl, ad esempio tramite l'aggiunta di nuove funzioni. Chiunque può creare un modulo e renderlo disponibile a tutti tramite il Comprehensive Perl Archive Network (<http://search.cpan.org>): il sito ne contiene migliaia, molti dei quali sono tra l'altro già inclusi nella distribuzione standard dell'interprete. Mentre creare un modulo richiede qualche conoscenza un po' più avanzata di Perl, il loro utilizzo è piuttosto semplice. Molti di essi a dire il vero sono oggetti, e quindi richiedono delle conoscenze minime di OOP (object oriented programming). In questa sede verranno fornite solo le formazioni indispensabili al loro uso, rinviando chiunque fosse interessato ad approfondire la conoscenza della programmazione OOP alla relativa documentazione. I moduli distribuiti con l'interprete Perl si distinguono in due principali categorie: quelli *pragmatici* e tutti gli altri.

## Moduli pragmatici

I moduli pragmatici sono dei moduli speciali, in quanto essi forniscono delle direttive all'interprete in modo che esso analizzi il codice in un determinato modo. Vediamo ad esempio l'effetto del modulo pragmatico *integer*:

```
use integer;
$a = 11/3;
```

Anzitutto si nota l'utilizzo di *use*, la direttiva che indica appunto l'inclusione di un modulo, il cui nome è passato come parametro. L'interprete verifica anzitutto se tale parametro indica un modulo pragmatico, e si comporta di conseguenza. Nel caso così non fosse, il modulo viene cercato nei path standard previsti dall'interprete. Se si desiderasse specificarne altri si può inserire, come prima istruzione *use*:

```
use lib '/usr/lib/mymodules';
```

In questo specifico caso i moduli verranno cercati anche in */usr/lib/mymodules*. Tra l'altro anche *lib* è un modulo pragmatico. Tornando ad *integer*, esso fa sì che tutte le operazioni matematiche vengano compiute su numeri interi, quindi *\$a* assume qui valore *3* anziché *3.66666666667*. È importante notare che questo modulo non ha nessun influsso sull'assegnazione dei valori alle variabili, ma solo sulle operazioni su di esse, quindi:

```
$a = 4.75;
```

assegna effettivamente *4.75* a *\$a*. I moduli pragmatici non sono molti, ma alcuni di essi tornano utili in specifici casi. Ad esempio *warnings* permette di abilitare o disabilitare la stampa sullo standard error dei warning, cioè gli errori non fatali, controllando anche quali tipi di warning vadano effettivamente visualizzati. I moduli pragmatici possono anche essere utilizzati solo su parti del proprio programma, e precisamente solo all'interno dei blocchi di codice nei quali si vuole abbiano effetto. Ad esempio:

```
$a = 11/3;
{
  use integer;
  $b = 11/3;
  print "$b\n";
}
print "$a\n";
```

Il primo valore stampato (*\$b*) è intero, mentre il secondo è un numero in virgola mobile. Una direttiva interessante è quella che permette di specificare la versione di Perl *minima* richiesta. Ad esempio:

```
use 5.8.0;
```

posto all'inizio del proprio programma fa sì che esso non venga eseguito in versioni dell'interprete antecedenti alla 5.8.0.

## Moduli non pragmatici

Più che moduli non pragmatici, questi andrebbero chiamati semplicemente moduli. Essi si distinguono in quelli standard e quelli non standard. La differenza tecnica è inesistente: la categorizzazione è dovuta al semplice fatto che i primi si trovano assieme all'interprete, mentre i secondi vanno cercati nel CPAN oppure in giro per Internet. Vediamo qualcuno di quelli standard, partendo dal vasto ed affascinante mondo della matematica:

```
use Math::Trig;
$ch = cosh($a);
```

Come si nota il modulo permette di chiamare la funzione `cosh()`, che calcola il coseno iperbolico. Allo stesso modo sono fornite funzioni per il calcolo di tangente, arcoseno, cosecante e via dicendo. Poniamo ora di voler utilizzare interi molto grandi, cioè al di sopra dell'intervallo che il nostro sistema operativo mette a disposizione (solitamente da -2147483647 a +2147483647). Ci viene incontro *Math::BigInt*:

```
use Math::BigInt;
$a = Math::BigInt->new("14598271981201015819041");
print ($a*3)."\n";
```

Questo codice crea, chiamando il *costruttore* dell'oggetto *Math::BigInt*, un numero intero grande (la grandezza è indefinita, il modulo non pone limiti) e lo memorizza in *\$a*. Il costruttore è *new()*: in maniera semplicistica potremmo definirlo una funzione che, a partire da una stringa, crea un'entità adatta a rappresentare un numero intero grande. Una volta creato, il numero (che in realtà è un oggetto, non una variabile scalare) può essere utilizzato come un intero qualsiasi, e quindi su di esso sono applicabili tutti gli operatori matematici. Lasciamo ora stare i numeri e passiamo a *File::Copy*, modulo non OOP che aggiunge a Perl le funzioni *copy()* e *move()*, le quali permettono appunto di copiare e spostare file:

```
use File::Copy;
# Copia Anna_Lee.mp3 della directory /mp3/DT
copy ("/mp3/Anna_Lee.mp3" "/mp3/DT/Anna_Lee.mp3");
# Copia Eileen.mp3 della directory /mp3/SkidRow
move ("/mp3/Eileen.mp3" "/mp3/SkidRow/Eileen.mp3");
```

Questo, per operazioni di copia o spostamento semplici, cioè che non sfruttino ricorsione o altre opzioni più avanzate, evita di dover chiamare un comando esterno con *system()* o con l'apostrofo inverso ( ` ), detto anche *backtick*). È anche possibile confrontare due file con *File::Compare*:

```
use File::Compare;
if (compare ("Eileen.mp3", "SkidRow-Eileen.mp3") )
{
    print "Sono uguali!";
} else {
    print "Sono diversi!";
}
```

Anche in questo caso il modulo mette semplicemente a disposizione una nuova funzione, senza che vengano sfruttati gli oggetti. Analizziamo ora il comodo *Text::Wrap*, che formatta automaticamente un testo in modo che rientri in un certo numero di colonne. Poniamo di avere il paragrafo riportato nella prima parte di **Riquadro 1** in *\$a* e di processarlo con il seguente codice:

```
use Text::Wrap;
$text::Wrap::columns = 30;
print wrap (" " x 5, " ", $a);
```

La seconda riga è di un certo interesse: essa assegna un valore ad una variabile interna *Text::Wrap*, ed in particolare a quella che determina il numero massimo di colonne per ogni riga. L'ultima linea di codice è quella che fa il lavoro vero e proprio, chiamando la funzione *wrap()*. I parametri sono, nell'ordine: margine sinistro della prima riga, margine sinistro di tutte le altre righe e variabile contenente il testo originale. Il risultato è visibile nella seconda parte di **Riquadro 1**.

La citazione kantiana prima della formattazione:

```
Due cose riempiono l'animo di ammirazione e riverenza sempre nuova e  
crescente: il cielo stellato sopra di me la legge morale dentro di me.
```

... e la stessa dopo essere stata trattata con `Text::Wrap`...

```
    Due cose riempiono  
l'animo di ammirazione e  
riverenza sempre nuova e  
crescente: il cielo stellato  
sopra di me la legge  
morale dentro di me.
```

Inclusi nella distribuzione, soprattutto se la versione è piuttosto recente, si possono trovare moltissimi altri moduli, sufficienti per tutte le operazioni più comuni.

## Conclusioni

Con l'analisi di alcune funzioni e di alcuni dei principali moduli si chiude questo piccolo corso di Perl. Gli argomenti da affrontare sarebbero ancora molti, a partire dalla programmazione object oriented di cui si parlava prima per finire alla creazione di moduli propri. Le informazioni date in queste dieci lezioni sono comunque sufficienti, o almeno mi auguro lo siano, a permettere a chi prima non sapeva nulla di Perl di svolgere le operazioni fondamentali. Gli argomenti più avanzati si possono apprendere da un buon libro, come quello indicato in bibliografia, oppure direttamente dalla completissima documentazione in linea. Con "in linea" intendo sia il materiale reperibile su Internet che quello incluso direttamente con l'interprete. La mia speranza è inoltre che questi articoli siano serviti a convincere qualcuno a provare Perl, ad incuriosirlo su questo linguaggio potente e moderno, ad indurlo a scoprirne le infinite sfaccettature, a considerarlo come un'alternativa, un modo di programmare e di pensare libero in un panorama dominato dalle soluzioni proprietarie.

# 11. Listati

## Listato 5.1

```
# Array bidimensionale
@persone = (
  [ "Michele", "Via San Francesco, 16" ],
  [ "Markus", "402, Cedars Drive", "1-800-400-1123"],
  [ "Annie", "40, Chester Lane"],
);

# Array di hash
@persone2 = (
  {
    "nome"      => "Michele",
    "indirizzo" => "Via San Francesco, 16",
  },
  {
    "nome"      => "Markus",
    "indirizzo" => "402, Cedars Drive",
    "telefono"  => "1-800-400-1123",
  },
  {
    "nome" => "Annie",
    "indirizzo" => "40, Chester Lane",
  },
);

# Hash di array
%persone3 = (
  "Italia" =>
    [ "Michele", "Via San Francesco, 16" ],
  "USA" =>
    [ "Markus", "402, Cedars Drive", "1-800-400-1123" ],
  "UK" =>
    [ "Annie", "40, Chester Lane"],
);

# Hash bidimensionale
%persone4 = (
  "Italia" => {
    "nome"      => "Michele",
    "indirizzo" => "Via San Francesco, 16",
  },
  "USA" => {
    "nome"      => "Markus",
    "indirizzo" => "402, Cedars Drive",
    "telefono"  => "1-800-400-1123",
  },
  "UK" => {
    "nome" => "Annie",
    "indirizzo" => "40, Chester Lane",
  },
);
```

## 12. Bibliografia

1. Larry Wall, Randal L. Schwartz - "*Programming Perl*", O'Reilly & Associates, Gennaio 1991
2. Larry Wall, Tom Christiansen, Randal L. Schwartz - "*Programming Perl (2<sup>nd</sup> edition)*", O'Reilly & Associates, Settembre 1996
3. Tom Christiansen, Nathan Torkington - "*Perl Cookbook*", O'Reilly & Associates, Settembre 1998
4. Ellen Siever, Stephen Spainhour, Nathan Patwardhan - "*Perl in a Nutshell*", O'Reilly & Associates, Gennaio 1999
5. Jon Orwant, Jarkko Hietaniemi, John Macdonald - "*Mastering algorithms with Perl*", O'Reilly & Associates, Agosto 1999
6. Larry Wall, Tom Christiansen, Jon Orwant - "*Programming Perl (3<sup>rd</sup> edition)*", O'Reilly & Associates, Luglio 2000