Tx0's Cammelling Perl!

Corso di Perl del LOA HackLab Milano

Smetti di programmare quando gli altri finiscono di capire il problema. E non chiamarlo problema, chiamalo "uhm, interessante..."! Al resto ci pensa Perl!

Indice delle Figure

Indice delle Tabelle

Indice

Capitolo 1.
Introduzione a Perl

-

Perl è stato creato da Larry Wall e molti altri programmatori *"across the Internet"*©. Lo scopo del linguaggio era aiutare la configurazione di un complesso sistema multipiattaforma della cui gestione Larry era stato incaricato. Serviva un tool con una buona programmabilità che consentisse il parsing dei file di configurazione e di log nella maniera più semplice possibile. Come tutte le cose belle, cresce, si sviluppa, si diffonde, si amplia in possiblità e sintassi. Diventa il linguaggio che qui presentiamo.

Mi interessa delimitare bene quello che effettivamente si copre in questo corso. Perl è un linguaggio immenso per complessità (nel senso buono) della sintassi e per possibilità offerte dalle librerie (moduli) che si possono reperire su Internet. Praticamente qualsiasi task di amministrazione, comunicazione, catalogazione, interazione, computazione e qwertyzzazione è già stato affrontato, sviscerato e risolto. Molti di questi moduli sono talmente di uso comune che sono entrati a far parte della *libreria standard* di Perl, ossia di quell'insieme di moduli che viene fornito con il codice sorgente dell'interprete da compilare.

In questo corso in realtà noi copriremo la sintassi ed i concetti di base, daremo uno sguardo alla programmazione orientata agli oggetti e ad alcune altre caratteristiche avanzate e studieremo alcuni moduli. Molti di questi tuttavia (e persino la maggior parte delle funzioni builtin) non saranno invece contemplate. Ad esempio la parte di networking a basso livello non è qui presentata. Il motivo di questa scelta è che il corpo di materiale incluso nel programma è a stento contenibile in 10 lezioni, aggiungere altri concetti comporterebbe un corso di lunghezza estenuante.

Questo non esclude che un riscontro di interesse verso materie specifiche (ad esempio la programmazione con i socket, l'interfacciamento ai database SQL, la creazione di interfaccie con Gtk, la scrittura di CGI con CGI.pm) non possano originare minicorsi a se stanti intesi come prolungamento del corso principale.

Per ora godetevi la corsa, poi fatemi sapere! Buon viaggio! ;-)

1. La filosofia di Perl

Perl è un linguaggio particolare, modellato sulla lingua inglese **parlata** e improntato alla snellezza. Mutua la maggior parte della sua sintassi da due linguaggi ben noti (almeno superficialmente) a qualsiasi utente UNIX: il C e gli script di shell (sh/csh).

E fin qui nulla di sconvolgente. Perché mai Perl sarebbe così particolare? Spiegarlo tutto qui in introduzione non è concepibile; è questo un feeling che si apprezza con la progressiva conoscenza del linguaggio. Quello che sin da ora posso anticiparvi (e non sono parole a vanvera, ne illazioni pronunciate da un *Microsoft Certified Solution Reseller*) è che Perl è in grado di capire quello che voi volete da lui. E per capire intendo proprio quel modo che hanno le persone di capire una frase anche quando questa non è totalmente univoca e lascerebbe un minimo margine di ambiguità. Perl sa novantanove volte su cento quello che gli state chiedendo. Insomma: una volta su cento potete anche programmare voi! ;-)

Certo, Perl non è un'intelligenza artificiale, non vi sto proponendo di chiaccherare con il

vostro interprete Perl: in fondo non è mica Emacs (vero Little John? ;-) Ma scoprirete mano a mano che la programmazione si fa più profonda e intricata che Perl è una vera e propria lingua, solo che è stata adattata un minimo per un calcolatore. E se ancora non mi credete, sappiate che esiste un concorso annuale di poesia scritta in Perl! Attenzione: non **CON** Perl! **IN** Perl, usando sintassi Perl e parole Perl per comporre un testo che è un vero programma Perl, ma che è anche una poesia! Fanatismo? Seguitemi e potrete giudicare da voi. Magari deciderete di diventare fanatici voi stessi!

2. Un linguaggio interpretato

Perl è nato come linguaggio interpretato, con tutti i vantaggi e gli svantaggi che questo comporta.

Vantaggi:

- Portabilità totale
- Evidente ed ineludibile disponibilità del codice sorgente
- Rapido ciclo di prgrammazione, esecuzione, debugging

Svantaggi:

- Maggior lentezza di esecuzione rispetto ai compilati
- Rischio di danni al sorgente

Intendiamoci, la lentezza ed i rischi per il codice non sono assolutamente aspetti allarmanti della questione. La lentezza è sicuramente più incisiva dell'equivalente programma scritto in C; per questo Perl non è adatto alla stesura di grossi programmi di calcolo che dovranno presumibilmente elaborare per mesi e mesi consecutivi stream di dati a ciclo continuo. Questo no.

Ma se il vostro programma conclude l'elaborazione in mezz'ora quando il C avrebbe impiegato 15 minuti, la differenza è abbondantemente compensata dal tempo che avete risparmiato in fase di scrittura del codice. Perl è infatti incredibilmente veloce tanto nella fase cosidetta di **prototyping**, quanto in quella di stesura effettiva. La prima fase è quel momento iniziale in cui il programma viene delineato, se ne realizza la struttura generale, si individuano le parti, le sezioni e i pezzi di codice da scrivere (è un po' come segnare con il carboncino la tela prima di dipingerla).

La fase successiva (quella di programmazione, quando i colori effettivamente finiscono sulla tela) è altrettanto veloce. Perl ha una sintassi incredibilemente compatta e rapida. Con pochi caratteri è possibile esprimere notevoli concetti.

3. Un linguaggio parlato

Dato che non mi avete creduto guardate qui:

Perl e la poesia Fig. 1

```
#!/usr/bin/perl

APPEAL:

listen (please, please);

    open yourself, wide;
       join (you, me),
    connect (us,together),

tell me.

do something if distressed;

        @dawn, dance;
        @evening, sing;
        read (books,$poems,stories) until peaceful;
        study if able;

        write me if-you-please;

sort your feelings, reset goals, seek (friends, family, anyone);

            do*not*die (like this)
            if sin abounds;

keys (hidden), open (locks, doors), tell secrets;
    do not, I-beg-you, close them, yet.

                              accept (yourself, changes),
                              bind (grief, despair);

    require truth, goodness if-you-will, each moment;

select (always), length(of-days)

# listen (a Perl poem)
# Sharon Hopkins
# rev. June 19, 1995
```

Per favore, non mi chiedete cosa faccia questo script. Quello che conta è cosa dice questo script.

Buono studio!

Inizio Capitolo Indice

Indice

Capitolo 2.
Come usare l'interprete perl

La prima cosa con la quale imparare a convivere è l'interprete perl. Perl è un linguaggio interpretato, questo significa il codice scritto non viene compilato una volta per tutte per essere poi eseguito su uno specifico tipo di processore, ma viene letto ogni volta e compilato "run time" ossia al momento dell'esecuzione per il processore sul quale si vuole eseguire lo script. Il programma che si occupa di questa operazione è noto come interprete.

L'interprete ha un percorso completo con il quale è possibile richiamarlo. Ad esempio: /usr/bin/perl. Nel resto del testo useremo sempre e comunque questa convenzione.

1. Codice da STDIN

Una prima possibilità è passare il nostro codice all'interprete dal canale aperto su STDIN (questo canale viene aperto insieme a quello di STDOUT e STDERR automaticamente all'avvio dell'interprete):

Inizio Capitolo Indice

Capitolo 3.

Quotatura ed operatori

-

1. Quotatura

Per quotatura si intende l'inclusione di informazioni all'interno di simboli per privare uno o più caratteri del loro significato speciali. L'esempio più immediato è l'inclusione di una stringa di testo contenente spazi dentro ad un paio di virgolette, come in "Stringa con caratteri di spazio". Perl offre comunque una serie di possibili opzioni per privare un numero più o meno esteso di caratteri dei loro significati particolari e per limitare l'utilizzo della quotatura singola attraverso il backslash (\).

Vediamo alcuni esempi di quotatura:

Esempi di quotatura — Fig. 1

```
 1: #!/usr/bin/perl
 2:
 3: #
 4: # Quotatura semplice [ interpola ]
 5: #
 6: $scalar = "12";
 7: $string = "\$scalar = $scalar"; # Vale -->$scalar = 12<--
 8:
 9: #
10: # Quotatura totale [ non interpola ]
11: #
12: $scalar = '12';
13: $string = '\$scalar = $scalar'; # Vale -->\$scalar = $scalar<--
```

Esiste inoltre un sistema parallelo di quoting che consente un più semplice (per quanto inusuale) commento dei caratteri:

Quote (forma generale e particolare) Tabella 1

Particolare	Generale	Significato	Interpolazione
''	q//	Letterale	No
""	qq//	Letterale	Si
``	qx//	Comando	Si
()	qw//	Lista di parole	No
//	m//	Pattern Match	Si
s///	s///	Sostituzione	Si
y///	tr///	"Traduzione"	No

Questo sistema consente di avvalersi dei benefici della quotatura senza dover abusare del backslash (\). Il carattere utilizzato come delimitatore può essere scelto a seconda di quale risulti più comodo a seconda della composizione del testo che si sta quotando. Facciamo un esempio. <FONT FACE="Arial,Helvetica">Prova</FONT> Questo pezzo di sorgente HTML contiene alcuni caratteri "scomodi" da quotare come le virgolette. Quindi è scomodo da quotare con le virgolette. Lo slash in </FONT> inoltre consiglierebbe di non usare la forma qq// per quotare ma di scegliere un carattere delimitatore differente. Con HTML in generale risulta molto conveniente il carattere pipe (|). Vediamo le possibili quotature:

Esempi comparati di quotatura Fig. 2

```
 1: #!/usr/bin/perl
 2:
 3: #
 4: # Quotiamo la stringa <FONT FACE="Arial">Prova</FONT>
 5: #
 6:
 7: #
 8: # Quotiamo con le virgolette
 9: #
10: $string = "<FONT FACE=\"Arial\">Prova</FONT>;
11:
12: #
13: # Sconveniente! Abbiamo dovuto commentare 2 caratteri con un \
14: # Proviamo con la forma generale: qq//
15: #
16: $string = qq/<FONT FACE="Arial">Prova<\/FONT>/;
17:
18: #
19: # Meglio! Ma abbiamo ancora uno slash commentato
20: # Cambiamo delimitatore:
```

```
21: #
22: $string = qq|<FONT FACE="Arial">Prova</FONT>|;
```

2. Aritmetici

Iniziiamo con gli operatori più immediati: gli operatori aritmetici.

Operatori aritmetici	Tabella 2
+	somma
-	sottrazione
*	moltiplicazione
/	divisione
%	modulo
**	elevamento a potenza

3. Stringhe

Operatori su stringa di testo	Tabella 3
.	concatenazione
x	ripetizione

Esempi con le stringhe — Fig. 3

```
 1: #!/usr/bin/perl
 2:
 3: #
 4: # Concatenazione
 5: #
 6: $string = "Ciao" . " " . 'Mondo';
 7: print $string . qq/\n/;
 8:
 9: #
10: # Ripetizione
11: #
12: $length = 10;
13: $string = "*" x $length;
14: print $string . "\n"; # Stamperà"**********\n"
```

4. Logici e comparativi

Operatori logici			Tabella 4
Bassa priorità	**Alta priorità**	**Nome**	**Risultato**
$a && $b	$a and $b	And	$a se $a è falso, altrimenti $b
$a || $b	$a or $b	Or	$a se $a è vero, altrimenti $b
!$a	not $a	Not	la negazione di $a
$a ^ $b	$a xor $b	Xor	quella delle due vera, oppure nessuno se entrambe sono vere o false

Operatori di comparazione		Tabella 5
Numeri	**Stringhe**	**Significato**
>	gt	Maggiore di
>=	ge	Maggiore o uguale a
<	lt	Minore di
<=	le	Minore o uguale a
==	eq	Uguale a
!=	ne	Disuguale
<=>	cmp	Comparazione con risultato segnato

L'ultimo operatore richiede un commento particolare. Il risultato ritornato varia fra -1, 0 e 1.
I tre numeri indicano (per così dire) quale dei tre simboli componenti l'operatore è vero.
Diciamo che stiamo testando $a <=> $b; avremo questa possibile casistica:

L'operatore <=>	Tabella 6
$a < $b	-1
$a = $b	0
$a > $b	1

Gli operatori su file	Tabella 7
-r	leggibile
-w	scrivibile
-x	eseguibile
-o	dell'utente
-e	esistente
-z	grande zero byte (caratteri)
-s	grande più di zero byte (caratteri)
-f	un file normale
-d	una directory
-l	un link
-S	un socket
-b	block file
-c	character file
-t	aperto su una tty
-u	SUID bit impostato
-g	GUID bit impostato
-T	file di testo
-B	file binario

Inizio Capitolo Indice

Capitolo 4.

I tipi di dati

-

1. Premessa: il concetto di variabile

In qualsiasi linguaggio di programmazione, una variabile è una posizione di memoria contrassegnata da un nome univoco all'interno della quale è possibile depositare un valore per poterlo successivamente recuperare o elaborare.

In Perl è possibile utilizzare nomi composti di lettere, numeri e underscore (_), incluse quelle combinazioni di caratteri che costituiscono parole riservate. Questo è dovuto al fatto che le variabili sono precedute da simboli speciali che le contraddistinguono come tali.

2. Bestiario

Perl è un linguaggio scarsamente tipizzato. Questo significa che non è prevista una rigida distinzione fra i vari tipi di dati che possono essere utilizzati. In Perl infatti uno numero intero coincide con la stringa testuale che lo rappresenta. Pertanto il valore 5 non differisce dalla stringa di testo "5". Perl all'occorrenza trasforma il numero in stringa e la stringa in numero in modo da poter utilizzare correttamente il dato nel contesto in cui lo si sta utilizzando. Se il contesto è una concatenazione di stringhe, Perl trasformerà il numero 5 nella stinga di testo "5", risparmiando un bel po' di lavoro al programmatore.

Inoltre Perl utilizza tre caratteri riservati per marcare il tipo di dato. Il dollaro ($) indica gli scalari (il dollaro in fondo assomiglia alla S di Scalar); la a commerciale (@) per gli array (che appunto iniziano per A) e il simbolo di percentuale (%) che nella mente degli autori ricorda la H di Hash. Questo sistema è stato mutuato dai linguaggi di scripting preesistenti come sh e csh.

Vediamo subito quali sono i tipi di dati accettati da Perl:

I tipi di dati accettati da Perl — Fig. 1

```
1: #!/usr/bin/perl
2:
3: #
4: # Scalari (numeri interi, reali, stringhe di testo, reference, ... )
5: #
6: $int = 5;
7: $float = 5.2;
8: $string = "Corso di Perl";
```

```
 9:
10: #
11: # Array ( liste ordinate monodimensionali di elementi )
12: #
13: @array = ( 1, 2, 3 );
14: $array[3] = 12;
15:
16: #
17: # Hash ( array accessibili per chiavi )
18: #
19: %hash = ( "auto", "ford", "modello", "fiesta" );
20: $hash{auto} = "opel";
21: $hash{modello} = "corsa";
```

3. Contesti

Perl non tipizza fortemente i dati; per contro riserva grande attenzione al contesto nel quale i dati sono elaborati. Il contesto può essere scalare o di lista. I tipi di Perl valutati in differenti contesti si comportano in maniera differente. Tracciamo un breve profilo dei differenti contesti e degli effetti che sortiscono sui differenti tipi di dati:

I contesti Fig. 2

```
 1: #!/usr/bin/perl
 2:
 3: #
 4: # Contesto scalare
 5: #
 6: $sc = @array; # Dimensione di @array
 7: $sc = ( 'elem1', 'elem2', 'elem3' ); # L'elemento più a destra
 8:
 9: #
10: # Contesto di lista
11: #
12: # NOTA: localtime restituisce i seguenti valori: secondi, minuti, ore,
13: # giorno del mese, mese, anno, giorno della settimana, giorno dell'anno,
14: # anno bisestile in list context, il numero di secondi trascorsi dal
15: # 1 gennaio 1970 in scalar context (perché da quella data?
16: # mai sentito parlare di UNIX?)
17: #
18:
( $sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst) = localtime();
19: $date = localtime();
```

Alla linea 6 'assegnamo' un array ad uno scalare. Il contesto quindi sarà scalare (il contesto è sempre dato dalla parte sinistra dell'assegnazione, dato che è qui che il valore in elaborazione viene memorizzato e quindi a questa parte deve conformarsi). Come si comporta il nostro array? Dentro $sc troveremo il numero di elementi dell'array.

Diversamente si comporta una lista generica di elementi dati. Per quanto potrebbe sembrare in tutto analoga ad un array, una lista in contesto scalare ritorna l'ultimo elemento. In questo caso quindi, alla linea 7, stiamo assegnando alla variabile $sc il valore "elem3".

Anche le funzioni ritornano valori differenti a seconda del contesto in cui sono utilizzate. Vediamo l'esempio della funzione localtime. Nella nota sono segnati i valori riportati in un contesto scalare. La notazione alla linea 18 assegna a ciascuna variabile il corrispettivo elemento ritornato dalla funzione. Questo esempio è utile anche per introdurre un'altro concetto: l'utilizzo delle liste come raggruppamento di variabili.

Nell'ultimo caso (linea 19) localtime sa di agire in un contesto scalare e quindi ritorna un valore scalare pari al numero di secondi trascorsi dall' 1/1/1970.

Facciamo un esempio notevole per capire la versatilità dei raggruppamenti di variabili in liste. Perl è uno dei pochi linguaggi (se non l'unico) in grado di scambiare il contenuto di due variabili senza ricorrere ad una variabile temporanea. Vediamo come:

Come scambiare il contenuto di due variabili — Fig. 3

```
1: #!/usr/bin/perl
2:
3: $var1 = 15;
4: $var2 = 30;
5:
6: ($var1, $var2) = ($var2, $var1);
7:
8: print "var1 = $var1, var2 = $var2\n";
```

La linea 6 *fa la magia.* Il risultato dell'ottava linea di codice sarà: "var1 = 30, var2 = 15". Avremo comunque modo di approfondire il concetto in seguito.

4. Gli Scalari

Gli scalari sono il tipo base di informazione gestita da Perl. Comprendono:

- Numeri interi
- Numeri reali
- Stringhe di testo
- Reference ad altre variabili

Escludendo per ora il concetto di **reference**, concentriamoci sui tipi più comuni. Vediamo alcuni esempi di valori scalari:

Esempi di valori scalari Fig. 4

```
 1: #!/usr/bin/perl
 2:
 3: $int     = 56;
 4: $float   = 12.786;
 5: $science = 6.02e23;
 6: $octal   = 0453;
 7: $exa     = 0xa347d;
 8: $binary  = 0b111; # Solo Perl versione 5.6 e successive
 9:
10: $string1 = "Corso di Perl";
11: $string2 = "LOA HackLab Milano";
12:
13: print "$int, $float, $string2";
```

Il risultato di questo codice sarà la stringa di testo "56, 12.786, LOA HackLab Milano".

5. Gli Array e le Hash

Come si può notare, gli unici tipi di dato che hanno una loro tipizzazione in Perl sono gli array e le hash. Questo è dovuto al fatto che, più che tipi di dati, questi sono collezioni di dati, strutture di dati e non dati propriamente detti. Gli elementi di un array (semplificando per un momento) sono sempre e comunque degli scalari. Ed è per questo che si usa $ prima di un elemento di un array anziché @. Facendo riferimento ad un elemento di un array si stà indicando uno scalare e non un array!!!

Il sistema di accesso agli elementi viene eseguito tramite **subscripting**. Ossia, indicata la struttura per nome, viene posto il numero dell'elemento desiderato fra parentesi quadre (nel caso di un array) oppure il nome della chiave desiderata fra parentesi graffe (nel caso di una hash). Nel conteggiare gli elementi di un array tenete sempre presente che il primo elemento è alla posizione 0 (zero).

Gli array inoltre godono di una comoda convenzione: la notazione $#array (dove array è ovviamente il nome dell'array) è la posizione occupata dall'ultimo elemento dell'array. Quindi, il codice seguente:

Notazione $#array — Fig. 5

```
1: #!/usr/bin/perl
2:
3: @days = (
4:   'lunedi', 'martedi', 'mercoledi',
5:   'giovedi', 'venerdi', 'sabato', 'domenica',
6: );
7:
8: print $#days;
```

ritornerà come valore "6" in quanto (partendo da zero) l'ultimo elemento di sette è il sesto.

Intanto, giocando con una particolare notazione, ci siamo però accorti che Perl è un linguaggio molto flessibile. Abbiamo dichiarato l'array su 4 righe (dalla 3 alla 6), dividendo su due righe i valori dentro l'array e dopo l'ultimo elemento abbiamo lasciato una virgola (nonostante ad esso non faccia seguito alcun elemento). L'ultima virgola è esplicitamente consentita da Perl per evitare che un domani, aggiungendo nuovi elementi all'array, ci si dimentichi la virgola alla fine di questa linea e per così poco si rischi un esaurimento da debugging.

A dimostrazione che Perl è sempre più flessibile di quello che vi aspettate, introduciamo ora una nuova notazione per creare hash. Anziché descriverla come una lista di elementi nella quale i dispari sono chiavi e i pari sono valori (vedi *"I tipi di dati accettati da Perl"*), possiamo usare l'operatore =>, come in Fig. 6:

L'operatore => e la crezione di hash — Fig. 6

```
1: #!/usr/bin/perl
2:
3: $hash = (
4:   marca   => 'opel'
5:   modello => 'corsa'
6:   porte   => 3
7: );
```

Oltre ad aggiungere indubbiamente chiarezza al codice, questo operatore ha anche il vantaggio di eliminare la necessità di racchiudere tra virgolette la chiave (posto che questa non contenga caratteri speciali, come lo spazio).

6. Filehandle

Un filehandle è un simbolo attraverso il quale si ha accesso al contenuto di un file. Più in generale, il concetto si estende anche ad un pipe di comandi e ad una connessione in rete. In Perl un filehandle è considerato un tipo di dato. I filehandle usano il loro namespace privato e non rischiano il conflitto con le altre variabili. Esistono tre filehandle che ogni programma Perl trova automaticamente aperti all'inizio dell'esecuzione: STDIN, STDOUT e STDERR (rispettivamente il canale di input, output ed output di errore). Vediamo alcuni esempi di uso:

Filehandle Fig. 7

```
 1: #!/usr/bin/perl
 2:
 3: print "Step 1\n";
 4: print STDOUT "Step 2\n";
 5:
 6: open( IN, "source.txt" );
 7: open( OUT, ">log.txt" );
 8:
 9: while ( $line = <IN> ) {
10:    print OUT "Next line: $line\n";
11: }
12:
13: close( IN );
14: close( OUT );
```

NOTA: la funzione print accetta come primo elemento opzionale un filehandle al quale scrivere; di default questo filehandle è STDOUT.

Questo codice produce due linee a "video" ('Step 1' e 'Step 2'); notate come l'uso di STDIN sia ridondante nella seconda linea. Nella terza linea viene usata la funzione open per aprire un file ("source.txt") ed associargli il filehandle IN. Il file source.txt è ora aperto in sola lettura. Nella quarta linea open apre un filehandle in scrittura verso il file "log.txt". Il carattere > prima del nome del file indica che sul file si vuole scrivere e non leggere. Inoltre l'uso di un singolo > indica che il file deve essere aperto dal byte zero, quindi azzerato se preesistente; l'uso di un doppio >> indica invece che il contenuto inviato al filehandle andrà appeso alla fine del file, dopo l'ultimo byte (come del resto perfettamente noto a tutti gli utenti UNIX e programmatori di shellscript).

Dalla quinta alla settima linea abbiamo un ciclo while che legge una linea alla volta il file source.txt e pone il contenuto del file (una linea alla volta) nella variabile $line. Per ciascuna linea letta scrive nel file "log.txt" la stringa "Next line: " seguita dalla linea appena letta. Terminato la lettura del file, i filehandle IN e OUT vengono chiusi.

Abbiamo introdotto dei concetti anzitempo: le funzioni open, close, l'istruzione while e dall'inizio usiamo la funzione print. Se la cosa avesse dovuto generare confusione, non

preoccupatevene: al momento non siete assolutamente tenuti a comprendere il significato preciso del codice che leggete. È sufficiente che ne comprendiate il meccanismo a grandi linee. Quello che conta è ora concentrarsi sui tipi e sulle variabili.

7. Reference

Tutti i linguaggi di programmazione hanno sviluppato un personale approccio all'uso della memoria. Molti includono il concetto di *valore che si riferisce ad una locazione di memoria*. C mette a disposizione i puntatori attraverso i quali si può interagire direttamente con la memoria del computer ed allocarne porzioni grandi a piacere per uno scopo. Java al contrario non consente di lavorare con la memoria in quanto ritiene la cosa troppo pericolosa rispetto i vantaggi che porta (che RMS possa perdonarvi! ;-).

Perl come al solito pensa per conto suo. Non consente di avere accesso diretto alla memoria anche perché non pone limite alcuno alla dimensione delle strutture dati e per conseguenza non richiede nemmeno di preoccuparsi esplicitamente dell'allocazione. In compenso consente di ottenere dei valori che rappresentano *l'indirizzo* di un dato simbolo per poterlo poi richiamare attraverso questo indirizzo. Per simbolo intendiamo qui una variabile di qualsiasi tipo, un filehandle e addirittura una subroutine.

Per ottenere una reference ad un simbolo è sufficiente farlo precedere da un backslash (\). Il significato di questa scrittura è uno scalare che contiene informazioni sul tipo di dato referenziato e il suo indirizzo in memoria. Vediamo:

Creazione di reference — Fig. 8

```
1: #!/usr/bin/perl
2:
3: $value = 10;
4: @array = ( 'elem1', 'elem2' );
5:
6: $ref_value = \$value;
7: $ref_array = \@array;
8: $ref_const = \3.14159;
```

Dopo aver assegnato due valori a due variabili, creiamo due variabili ($ref_value e $ref_array) che contengono una reference alle due variabili. Per avere un'idea di come siano conformate le reference possiamo stampare i valori contenuti in $ref_value e $ref_array (che non sono 10 e 'elem1', 'elem2', bensì : SCALAR(0x80d1fdc) e ARRAY(0x80d2048)). Notate come nella prima parte sia indicato il tipo di dato referenziato mentre la seconda parte è l'indirizzo vero e proprio di memoria al quale si trova il dato referenziato.

In ultimo notate come sia possibile creare reference anche a costanti, stringhe di testo e dati generici (linea 8).

Un'altra possibilità interessante viene da questa sintassi per la creazione di array e hash cosiddetti anonimi:

Creazione di array ed hash anonimi Fig. 9

```
1: #!/usr/bin/perl
2:
3: $array_ref = [ 'elem1', 'elem2', 'elemA', 'elemB' ];
4: $hash_ref = {
5:   key1 => 'value1',
6:   key2 => 'value2',
7: };
```

L'utilizzo delle parentesi quadre e graffe costruisce (rispettivamente) un'array o una hash senza associarle ad alcun nome di variabile, mentre ne ritorna semplicemente un reference. Sta al programmatore assegnare questo reference dentro una variabile.

Inizio Capitolo Indice
Indice

Capitolo 5.

Costrutti e cicli

In questo capitolo facciamo rientrare nella categoria dei costrutti tutte le keyword Perl che servono a generare un ciclo di iterazione o a eseguire codice in base al verificarsi di alcune condizioni.

1. Vero o falso?

Prima di iniziare a valutare come fare test con Perl vale decisamente la pena di descrivere cosa è vero e cosa è falso.

0 è falso, 1 è vero. E fin qui nulla di sconvolgente.

Una qualsiasi variabile non definita è falsa.

Una stringa non nulla è vera, una stringa nulla (es. "") è falsa. Quindi se 0 è falso, "0" invece **è vero** in quanto stringa di testo non nulla. Occorre quindi prestare molta attenzione al contesto nel quale i dati vengono valutati. Ad esempio: sappiamo che 0 è falso. Ma sarà vero o falso: `$test = "" . 0`? In questo caso la stringa è nulla, quindi il risultato è la **cifra** 0, quindi falso!

2. if e unless

if è il costrutto più fondamentale. Serve ad eseguire un arbitrario blocco di codice in base al verificarsi (o meno) di alcune condizioni. Vediamo subito la sintassi minima di if:

Sintassi minima di if — Fig. 1

```
 1: #!/usr/bin/perl
 2:
 3: #
 4: # Sintassi minima di if
 5: #
 6: #if ( CONDIZIONE ) {
 7: #  [ ... blocco di codice ... ]
 8: #}
 9:
10:  $int = 5;
11:
12:  if ( $int == 5 ) {
13:    print "\$int vale 5";
14:  }
```

Ovviamente è un esempio stupido, in quanto abbiamo la sicurezza che $int valga 5, ma non temete: stiamo per espandere l'esempio per dargli un minimo di senso!

Sintassi completa di if — Fig. 2

```
 1: #!/usr/bin/perl
 2:
 3: @values = (5,6,7,8);
 4: $int = $values[ rand(4) ];
 5:
 6: if ( $int == 5 ) {
 7:   print "\$int vale 5";
 8: } elsif ( $int == 6 ) {
 9:   print "\$int vale 6";
10: } else {
11:   print "\$int ha un altro valore!";
12: }
```

Cosa è cambiato? Prima di tutto il valore di $int ora viene scelto a caso fra i valori di @values (per i curiosi, rand genera un valore a caso fra 0 e il limite passato come argomento; quindi $int = $values[rand(4)]; setta $int in base al valore contenuto nella posizione rand(4)).

Ma la cosa fondamentale è che il nostro costrutto if ora ha ragione di esistere. Alla linea 6 si verifica che $int valga 5, alla riga 8 (con elsif) si verifica che valga 6, alla 10 si decide che nessuno dei valori precedentemente verificati sussite e quindi si sceglie una via alternativa. Lanciando ripetutamente lo script, esso darà risultati differenti ad ogni esecuzione.

else serve a scegliere una via di default (ossia una via da usare quando nessun'altra è praticabile). elsif è un ibrido fra else ed if, consente di tracciare un'altra via ma solo se il test di questa via è stato verificato.

if ha una negazione naturale in unless che vale (né più né meno) if not. Quindi if (not $int) è identico a unless ($int). Vedremo poi con i modificatori di comando come unless assuma maggiore significato.

3. while e until

Il costrutto while serve a ripetere un blocco di istruzioni finchè la condizione valutata da while è vera. Vediamo un esempio:

Sintassi di while — Fig. 3

```
1: #!/usr/bin/perl
2:
3: $int = 0;
4: while ( $int < 10 ) {
5:   print "\$int vale $int\n";
6:   $int++;
7: }
```

In questo esempio l'istruzione da valutare e' che $int sia minore di 10. All'interno del blocco di codice la variabile viene incrementata di una unità ad ogni esecuzione, con l'istruzione $int++. In questo modo, quando $int == 10 la condizione non è più valida e while termina.

Esistono due keyword in grado di modificare il corso di un ciclo while: sono last e next. next interrompe l'esecuzione del blocco di codice e salta direttamente *alla successiva iterazione*. last blocca l'esecuzione del blocco di codice ed esce dal ciclo while.

Un metodo più originale per scrivere il precedente ciclo while può essere il seguente:

while con last — Fig. 4

```
 1: #!/usr/bin/perl
 2:
 3: $int = 0;
 4: while ( 1 ) {
 5:   print "\$int vale $int\n";
 6:   $int++;
 7:   if ( $int >= 10 ) {
 8:     last;
 9:   }
10: }
```

Vediamo invece un uso serio di next! Diciamo che vogliamo modificare il codice dell'esempio precedente in modo che non stampi la riga in caso $int valga 5:

Uso di next — Fig. 5

```
 1: #!/usr/bin/perl
 2:
 3: $int = 0;
 4: while ( $int < 10 ) {
 5:   if ( $int == 5 ) {
 6:     $int++;
 7:     next;
 8:   }
 9:   print "\$int vale $int\n";
10:   $int++;
11: }
```

In questo modo quando il ciclo esegue l'istruzione condizionale, se $int vale 5 viene incrementata e subito dopo il blocco di codice viene interrotto e si passa alla successiva iterazione del while con $int che vale 6.

Questa soluzione è sicuramente valida ma esiste una possibilità sicuramente più elegante: continue. Mediante l'uso di questa keyword è possibile specificare un secondo blocco di codice da eseguire alla fine del primo blocco. L'esecuzione avviene anche in caso di chiamata a next. Vediamo come:

La keyword continue — Fig. 6

```
 1: #!/usr/bin/perl
 2:
 3: $int = 0;
 4: while ( $int < 10 ) {
 5:   if ( $int == 5 ) {
 6:     next;
 7:   }
 8:   print "\$int vale $int\n";
 9: } continue {
10:   $int++;
11: }
```

In questo secondo esempio, anche quando il blocco di codice esegue il next, la variabile $int sarà comunque incrementata di una unità in quanto l'istruzione di incremento si trova nel blocco specificato da continue. Esiste in Perl la possibilità di usare un keyword di significato opposto a while che ne è una completa negazione. Questa keyword è until. Tramite until il codice precedente si può riscrivere come:

L'uso di until — Fig. 7

```
 1: #!/usr/bin/perl
 2:
 3: $int = 0;
 4:
 5: #
 6: # ...equivalente a
 7: #
 8: #   while ( not $int >= 10 )
 9: #
10: # oppure a
11: #
12: #   while ( $int < 10 )
13: #
14: until ( $int >= 10 ) {
15:   if ( $int == 5 ) {
16:     next;
17:   }
18:   print "\$int vale $int\n";
19: } continue {
20:   $int++;
21: }
```

4. for e foreach

Il costrutto for server ad iterare un blocco di codice variando il contenuto di una variabile in un insieme di valori. Vediamo subito il primo esempio:

Il ciclo for — Fig. 8

```
1: #!/usr/bin/perl
2:
3: for ( $int = 0; $int < 10; $int++ ) {
4:   print "\$int vale $int\n";
5: }
```

Avrete già capito che questo codice è l'equivalente del codice con while scritto negli esempi precedenti.

Anche nei cicli for valgono le keyword next e last. Anche qui possiamo quindi includere una condizione per evitare che venga stampata la linea per $int pari a 5:

Uso di next nei cicli for — Fig. 9

```
 1: #!/usr/bin/perl
 2:
 3: #
 4: # Sintassi di for:
 5: # for ( INIT;     TEST;      INCR ) {
 6: #   blocco di codice;
 7: # }
 8: #
 9: for ( $int = 0; $int < 10; $int++ ) {
10:   if ( $int == 5 ) {
11:     next;
12:   }
13:   print "\$int vale $int\n";
14: }
```

In questo caso non abbiamo bisogno di usare tecniche particolari per assicurarci che $int sia incrementata anche in caso di last in quanto l'istruzione di incremento è inclusa nella dichiarazione iniziale del for. Ovviamente la prima clausola inizializza la variabile ($int = 0;) mentre la seconda clausola è quella da verificare ad ogni iterazione ($int < 10;).

La seconda forma di for enumera esplicitamente i valori sui quali la variabile viene iterata. Per questa seconda versione si usa la forma foreach che esprime meglio il

concetto di iterazione per ciascun elemento elencato. In realtà i programmatori fluenti in $Perl usano scrivere comunque la forma for in quanto più compatta:

foreach Fig. 10

```
1: #!/usr/bin/perl
2:
3: foreach $var ( 1, 2, 3, "abc", "xyz", 3.14 ) {
4:   print "A questo giro \$var contiene $var\n";
5: }
```

Notate come in questo caso la specifica dei valori da iterare sia necessariamente più lunga ma consenta di includere valori differenti (stringhe di testo e numeri assieme), cosa non possibile con la prima sintassi. Questa sintassi è particolarmente utile per iterare sui valori di un array:

Iterazione sui valori di un array Fig. 11

```
1: #!/usr/bin/perl
2:
3: @array = ( 1, 2, 3, "abc", "Tx0's Cammelling Perl" );
4: for $var ( @array ) {
5:   print "A questo giro \$var contiene $var\n";
6: }
```

5. I modificatori di comando

Le istruzioni specificate in questo capitolo possono essere usate in coda ad una istruzione come "modificatori di comando", aggiungendo un nuovo significato (condizionale o iterativo) al comando. Ad esempio se vogliamo che una istruzione sia eseguita solo se una condizione è verificata possiamo usare uno dei due seguenti esempi:

if standard e come "modificatore di comando" Fig. 12

```
1: #!/usr/bin/perl
2:
3: $int = 23;
4:
5: if ( $int == 23 ) {
6:   print '$int vale 23!';
7: }
8:
9: print '$int vale 23!' if $int == 23;
```

Notate come if sia usato in coda al comando print nel secondo esempio ottenendo una sintassi decidamente più compatta, elegante e leggibile (suona più o meno come "stampa questo se $int vale 23"). Ancora più interessante risulta il for come modificatore:

for come modificatore — Fig. 13

```
1: #!/usr/bin/perl
2:
3: @elementi = ( 'abc', q/elemento 2/, qw/uno due tre/ );
4: print for @elementi;
```

Con una istruzione incredibilmente compatta abbiamo mostrato (ok, magari un po' appiccicati) l'intero contenuto di un array.

6. Label di riferimento

È possibile contrassegnare un ciclo for o while ma anche un qualsiasi blocco di codice incluso fa una coppia di parentesi graffe con una etichetta. Queste label servono a fare riferimento al costrutto in maniera inequivocabile con le istruzioni next e last. Questa opportunità risulta molto utile in caso di costrutti nidificati (ossia un ciclo all'interno di un altro ciclo) in quanto una istruzione come next si riferisce normalmente al ciclo più prossimo e non sarebbe altrimenti possibile interromperne uno più esterno.

Facciamo l'esempio di una coppia di cicli for nidificati per esplorare una matrice bidimensionale:

Due cicli for nidificati — Fig. 14

```
1: #!/usr/bin/perl
2:
3: ESTERNO: for $x ( 1, 2, 3, 4 ) {
4:   for $y ( 10, 20, 30 ) {
5:     last ESTERNO if $x == 4;
6:     print "X x Y: ", $x * $y;
7:   }
8: }
```

Il comando last si riferisce in questo caso al comando for marcato come ESTERNO e non a quello più vicino (ossia il più interno).

7. Come si scrive uno schema switch/case

In Perl non esiste un costrutto particolare per descrivere una struttura a scelta multipla. È sicuramente possibile scrivere un'elenco di if concatenati (orrore!!), ma questa non è la

soluzione più semplice. Impostando la variabile $_ sul valore da confrontare si può usare un semplice and logico (&&) in questo modo:

Scrittura di un costrutto switch/case Fig. 15

```
 1: #!/usr/bin/perl
 2:
 3: $_ = 'abc';
 4:
 5: SWITCH: {
 6:   /abc/ && do {
 7:     print "Le prime tre lettere dell'alfabeto\n";
 8:     last SWITCH;
 9:   };
10:   /uvz/ && do {
11:     print "Le ultime tre lettere dell'alfabeto\n";
12:     last SWITCH;
13:   };
14:   /nomatch/ && do {
15:     print "Sorry, non ho trovato una corrispondenza adeguata!\n";
16:     last SWITCH;
17:   };
18: }
```

Notate come l'uso della label SWITCH consenta il salto della restante parte di test se uno trova una corrispondenza valida. Inoltre l'uso della label aiuta a specificare agli occhi del lettore che si tratta dell'equivalente di un costrutto switch e a capire a cosa il last faccia riferimento.

Inizio Capitolo Indice

Capitolo 6.
Subroutine

-

1. Dichiarazione di subroutine

Esistono differenti modi per dichiarare una subroutine:

Dichiarazioni di subroutine	Tabella 1
sub NOME { corpo }	Dichiara la subroutine NOME
sub NOME PROTO { corpo }	Dichiara la subroutine NOME con i parametri PROTO
sub [PROTO] { corpo }	Dichiara la subroutine ANONIMA

Il primo tipo di dichiarazione è il più semplice e quindi quello che scegliamo come punto di partenza nella nostra trattazione. Il NOME della subroutine è il simbolo con il quale ne richiameremo l'esecuzione, il corpo (sempre racchiuso fra parentesi graffe, anche se consta di una sola linea) è l'insieme di istruzioni che la subroutine rappresenta.

Vediamo un esempio:

Esempio di dichiarazione ed uso di subroutine — Fig. 1

```
1: #!/usr/bin/perl
2:
3: sub prova {
4:   print "Corso di Perl\n";
5: }
6:
7: prova;
```

La dichiarazione alla linea 3 crea una subroutine di nome prova che contiene una singola istruzione (print "Corso di Perl\n";). Alla linea 7 la subroutine viene richiamata. L'output del programma è facilmente prevedibile.

Fermiamoci a riflettere un momento su come questa subroutine è stata chiamata. Il suo solo nome è bastato perché l'interprete riconoscesse in quel punto una chiamata ad un blocco di codice definito altrove. Tuttavia: è sempre così ?

La risposta è NO! In questo caso la subroutine prova non accetta parametri e quindi costituisce un caso di chiamata dei più semplici. Ma se essa accettasse dei parametri si renderebbe necessario inviarli in maniera non ambigua. (Ricordate che in Perl le parentesi attorno agli argomenti di una sub non sono obbligatorie). Ammettiamo ad esempio che prova venga chiamata in questa maniera: `my $int = prova +5;`. Il `+5` costituisce parametro per la sub prova oppure è un elemento della linea di codice che dovrà essere sommato al *risultato* di prova? Basta una semplice prova per accorgersi che il compilatore, non sapendo come gestire il valore `+5` lo ha passato come parametro alla subroutine. Come dire a Perl che non è un argomento della subroutine? Semplice! Basta: `my $int = prova() + 5;`! In questo modo la lista di argomenti di prova è stata dichiarata una volta per tutte come una **lista nulla**.

L'aggiunta delle parentesi tonde caratterizza una parola come una subroutine. Esiste però un simbolo speciale dedicato alle subroutine: `&`. Qualsiasi parola preceduta da esso è da considerarsi nome di una subroutine. Perché la necessità di un simbolo speciale per le sub? Per quanto in realtà non strettamente necessario nella più canonica delle programmazioni strutturate (le tonde, abbiamo visto, già bastano a dichiarare una sub), questo simbolo gioca un ruolo importante nella creazione di reference di subroutine. Ma questo lo affronteremo in seguito.

2. La gestione degli argomenti

Gli argomenti passati alla sub sono contenuti nell'array `@_`. Ammettiamo dunque che:

Gestione degli argomenti di una subroutine — Fig. 2

```
 1: #!/usr/bin/perl
 2:
 3: sub prova {
 4:   my $first = $_[0];
 5:   my $second = $_[1];
 6:
 7:   print join ", ", $first, $second;
 8: }
 9:
10: &prova('elem1', 'elem2');
```

Notate come all'interno della sub, i due argomenti passati siano nelle rispettive posizioni di `@_`.

Occorre tuttavia fare molta attenzione alla logica con la quale si passano i parametri. I parametri passati alla subroutine vengono infatti fusi ed "appiattiti" in un unico array. Non è pertanto possibile recuperare due array passati ad una sub separatamente. Vediamo in dettaglio:

Errato passaggio di parametri — Fig. 3

```
 1: #!/usr/bin/perl
 2:
 3: sub prova {
 4:   my ( @ar1, @ar2 ) = @_;
 5:   print " --> @ar1\n";
 6:   print " --> @ar2\n";
 7: }
 8:
 9: my @a = (1,2,3);
10: my @b = (6,7,8);
11:
12: prova(@a, @b);
```

Nella sub, @ar1 ingloba tutti i parametri passati, divenendo così (1,2,3,6,7,8), mentre @ar2 è assegnato nullo. Come si risolve questo problema? Passando i parametri per reference!

Corretto passaggio di parametri — Fig. 4

```
 1: #!/usr/bin/perl
 2:
 3: sub prova {
 4:   my @ar1 = @{ $_[0] };
 5:   my @ar2 = @{ $_[1] };
 6:   print " --> @ar1\n";
 7:   print " --> @ar2\n";
 8: }
 9:
10: my @a = (1,2,3);
11: my @b = (6,7,8);
12:
13: prova(\@a, \@b);
```

Notate come la chiamata alla sub passi i valori per reference (usando il backslash davanti). Notate anche come i parametri passati siano degli scalari (reference), bisognosi quindi di essere convertiti (*typecast*) esplicitamente in array (linee 4 e 5).

Lo stesso ragionamento deve essere tenuto per i valori ritornati dalla sub. I valori in uscita sono appiattiti in una sola lista passata come argomento a return.

Un metodo molto comodo per gestire gli argomenti in ingresso è la funzione builtin

shift. Questa estrae il primo elemento da un array e lo ritorna. Iterate chiamate a shift riducono progressivamente un array fino a svuotarlo. Vediamo come usarla:

Gestione dei parametri con shift — Fig. 5

```
 1: #!/usr/bin/perl
 2:
 3: sub prova {
 4:   my @array = @{ shift() };
 5:   my $int = shift;
 6:
 7:   [ ... ]
 8:
 9: }
10:
11: &prova( \@array, $int );
```

Per ricostruire il primo array passato, prova prima chiama shift (essendo che non le sono forniti valori, questa opera per default su @_). shift recupera uno scalare che è un reference all'array passato. Il typecast esplicito (@{}) che avviene intorno a shift ne trasforma il risultato nell'array richiesto. Infine questo array viene assegnato a @array. Per il secondo parametro è ancora più semplice: essendo uno scalare non richiede typecast esplicito e shift può anche essere chiamata senza parentesi tonde. Attenzione perché questo è un punto che facilmente dà problemi.

Se chiamate shift senza parentesi tonde (o senza & davanti) Perl sarà confuso circa l'interpretazione da dare a shift. Deve intendersi come una chiamata a funzione o come una parola nuda deprecabilmente scritta senza nessun tipo di quotatura? Perl ricade sul primo caso normalmente, generando però un warning; cosa che viene eliminata dalle tonde in quanto queste tolgono ambiguità all'espressione.

3. Subroutine anonime

Una subroutine può anche non avere un nome! Non è un paradosso!! Il costrutto sub { .... } restituisce un reference alla subroutine. Quindi my $func = sub { print "Hello!\n"; } dichiara una sub richiamabile come &$func.

Una applicazione interessante di questa tecnica è una possibile alternativa al metodo per costruire un costrutto switch.

Emulazione di switch con sub anonime Fig. 6

```
 1: #!/usr/bin/perl
 2:
 3: my %h = (
 4:   valore1 => sub {
 5:     print "valore1\n";
 6:   },
 7:
 8:   valore2 => sub {
 9:     print "valore2\n";
10:   },
11: );
12:
13: my $valore = shift();
14: if ( defined $h{$valore} ) {
15:   &{$h{$valore}};
16: } else {
17:   print "Valore non trovato!\n";
18: }
```

Ciascun elemento della hash contiene una subroutine anonima da eseguire. La sintassi $h{$valore} ritorna un reference a subroutine. Quindi &{$h{$valore}} esegue la subroutine associata. In questo modo (e senza usare un singolo costrutto logico di test (if oppure &&) avete eseguito uno switch!

4. I prototipi

I prototipi servono a descrivere in maniera rigorosa quanti argomenti (e di quale tipo) la sub accetta. Ad esempio sub prova ($$) { ... } specifica che prova accetta solo due parametri scalari ($$). Il vantaggio della prototipizzazione delle sub è che è possibile chiamarle senza parentesi tonde senza che Perl incorra in problemi di interpretazione.

Ci sono da fare alcune considerazioni sui prototipi. Prima di tutto è bene pensare cosa è giusto prototipizzare. Una subroutine scritta due anni fà non è forse il miglior candidato alla prototipizzazione. Il software scritto fino ad ora che usa quella subroutine potrebbe avere dei problemi con la nuova prototipizzazione. È meglio prototipizzare solo le nuove subroutine.

Secondo: è necessario ragionare bene sui prototipi per non scrivere cose insensate. Per indicare un array o una hash intesi come reference è necessario prependere il simbolo con un backslash. C'è notevole differenza fra \@ e @: il primo è un array, il secondo è una lista che "consuma" tutti i parmetri rimanenti. Quindi se dovete prototipizzare una sub come func @array, $scalar scrivete sub func (\@$) { ... } come prototipo.

5. Pseudo espansione della sintassi attraverso l'uso dei prototipi

L'uso dei prototipi consente una 'pseudo' espansione della sintassi di Perl, con l'aggiunta di nuove parole chiave. Vediamo come:

Pseudo espansione della sintassi con i prototipi — Fig. 7

```
 1: #!/usr/bin/perl
 2:
 3: sub try (&$) {
 4:   my ( $try, $catch ) = @_;
 5:   eval { &$try };
 6:   if ( $@ ) {
 7:     local $_ = $@;
 8:     &$catch;
 9:   }
10: }
11:
12: sub catch (&) {
13:   return $_[0];
14: }
15:
16: try {
17:   die "Muoio!";
18: } catch {
19:   /Muoio!/ && print "il programma è morto!\n";
20: }
```

Come funziona? L'uso dei prototipi di funzione consente di chiamare una sub senza usare il simbolo iniziale di funzione &, e senza usare le parentesi tonde. In più Perl sa quanti parametri passare alla subroutine. Quindi in fase di esecuzione, Perl chiama try. Questa richiede un blocco di codice & e uno scalare generico ($). Quindi passa a try il blocco di codice che segue e lo scalare "che dovrebbe seguire" (ma che non segue). Perl si trova catch, subroutine dichiarata. Quindi invoca catch per recuperarne lo scalare richiesto. Questa ritorna come scalare il blocco di codice che le è stato passato.

A questo punto try può essere eseguita. Viene valutato il blocco di codice alla linea 5; se $@ (che contiene l'eventuale errore generato dall'esecuzione dell'ultimo eval) contiene qualcosa, allora esegue catch impostando una copia locale di $_ con il contenuto di $@.

In realtà il prototipo di try è stato dichiarato come (&$) solo per poter consentire l'introduzione di catch fra i due blocchi di codice. L'eliminazione di catch si potrebbe ottenere semplicemente con il prototipo sub try (&&) { .... }, il quale non richiede catch nel mezzo.

Inizio Capitolo Indice

Capitolo 7.
Scoping, Name Spaces, Packages e Moduli

-

1. Lo scope (campo di visibilità) delle variabili (my, local)

Lo scope di una variabile è il campo di visibilità all'interno del quale la variabile è raggiungibile (modificabile, risolvibile, interpolabile, ditela un po' come vi torna più congeniale).

Prendiamo il caso più semplice per fare un primo esempio. Ammettiamo di avere un programma di n righe di codice, all'interno del quale viene utilizzata una variabile. Diciamo anche che in questo programma non ci sono costrutti if, while, for e loro modificazioni. La visibilità della variabile in questione sarà estesa dalla riga di codice all'interno della quale viene definita, sino all'ultima riga del programma. Vediamo un esempio:

Primo esempio di Scoping — Fig. 1

```
1: #!/usr/bin/perl
2:
3: #
4: # Definiamo la variabile
5: #
6: $int = 5;
7:
8: print $int;
9: exit;
```

In questo caso è (forse) evidente che la visibilità della variabile abbia l'estensione citata in precedenza. Complichiamo un po' le cose:

Scoping dentro blocchi — Fig. 2

```
1: #!/usr/bin/perl
2:
3:
4: $int = 10;
5:
6: for ( $int = 0; $int < 5; $int++ ) {
7:   print "Dentro il ciclo \$int vale $int\n";
8: }
9:
10: print "Fuori dal ciclo \$int vale $int\n";
```

In questo caso lo scope della variabile $int si estende ancora a tutto lo script. Cosa avviene all'esecuzione dello script?

```
Dentro il ciclo $int vale 0
Dentro il ciclo $int vale 1
Dentro il ciclo $int vale 2
Dentro il ciclo $int vale 3
Dentro il ciclo $int vale 4
Fuori dal ciclo $int vale 5
```

Modifichiamo un minimo l'esempio precedente...

Limitare lo scoping — Fig. 3

```
1:  #!/usr/bin/perl
2:
3:
4:  $int = 10;
5:
6:  for ( my $int = 0; $int < 5; $int++ ) {
7:    print "Dentro il ciclo \$int vale $int\n";
8:  }
9:
10:  print "Fuori dal ciclo \$int vale $int\n";
```

...e vediamo cosa succede all'esecuzione dello script:

```
Dentro il ciclo $int vale 0
Dentro il ciclo $int vale 1
Dentro il ciclo $int vale 2
Dentro il ciclo $int vale 3
Dentro il ciclo $int vale 4
Fuori dal ciclo $int vale 10
```

Attenzione! Fuori dal ciclo $int vale 10!! Perché? Se fate attenzione noterete che abbiamo introdotto prima della variabile $int una parola chiave: my. Questo qualificatore di variabile consente di specificare variabili *"localizzate"*. Questo genere di variabili ha scope ridotto al più ristretto blocco di codice all'interno del quale la variabile è stata dichiarata. In questo caso particolare, lo scope di $int è ridotto alla durata del ciclo for. All'uscita dal ciclo for la variabile $int dichiaratavi scompare, riportando, per così dire, alla luce la variabile $int dichiarata in precedenza.

Una ulteriore proprietà del modificatore my è rendere la variabile estranea a qualsiasi Name Space (vedi il paragrafo successivo dedicato).

Contraltare di my è invece local, modificatore che duplica una variabile, assegnandole scoping temporaneo, relativo al campo rispettivo, deciso secondo le medesime regole di my. La differenza consiste nel fatto che local crea una copia della variabile, con medesimo nome, ma valore dissociato da quello della variabile precedentemente dichiarata.

Ad esempio è possibile eseguire il seguente codice:

Esempio di variabile local — Fig. 4

```
 1: #!/usr/bin/perl
 2:
 3: #
 4: # Definiamo l'array
 5: #
 6: @array = (1,2,3,4,5);
 7:
 8: #
 9: # Iteriamo su una copia locale dell'array
10: #
11: for ( my $c = 0; $c < @array; $c++ ) {
12:   local @array = @array;
13:   $array[$c]++;
14:   print qq/\$array[$c] incrementato vale $array[$c]\n/;
15: }
```

Il significato del ciclo for può essere parafrasato come: *"per $c che varia da 0 alla dimensione di @array (quindi finché $c è minore di 4), incrementando ad ogni iterazione di 1"*. Il blocco di codice eseguito all'interno effettua una copia di @array in un suo "clone" locale. Quindi incrementa il valore interessato di una unità e infine stampa il messaggio quotato.

2. Il concetto di Package e di NameSpace

La filosofia ufficiale di Perl è che non dovete entrare dove non è consentito, *non perché ci sono le sbarre alle finestre ma perché sapete di non doverlo fare!*

Per questo Perl provvede i Packages, ossia spazi che delimitano la visibilità di un simbolo ad un determinato ambito. Un *NameSpace* è letteralmente uno spazio letterale che caratterizza la posizione di un simbolo nella tabella dei simboli. (*Ricordiamo che per simbolo si intende qualsiasi "elemento atomico" dichiarato, comprendendo variabili, subroutine e filehandle*). In Perl il concetto di NameSpace e di Package coincidono. Quindi di quì in avanti parleremo di Packages (termine più perlish).

La struttura dei Packages è gerarchica (esattamente come la struttura delle directory su un filesystem). Ciascun livello è separato da un simbolo convenzionale. Originariamente questo simbolo era l'apostrofo ('), richiamando il fatto che un Package "appartiene" al Package superiore. (In Tools'Editors, Editors appartiene, nel senso di "è parte di", Tools). Successivamente il simbolo è stato sostituito da ::, conforme alla tradizione C. Quindi Tools'Editors diventa Tools::Editors. Tenete presente che la vecchia sintassi è ancora supportata per compatibilità; questo può portare ad errori subdoli, come print "This is $owner's car", in cui la variabile è $owner's, ossia la variabile s nel Package owner. Per fortuna affligge più gli anglofoni che i latini! ;-)

La dichiarazione di un Package avviene tramite la keyword package. Se all'inizio di un

file appare package Prova; tutto il contenuto del file farà parte del package Prova sino alla fine del file oppure sino ad una successiva dichiarazione di package, che attribuisce a questo secondo nuovo pacchetto il contenuto del file di lì in avanti!

Esiste un package speciale: main il quale include qualsiasi altro package. Incluso se stesso. Quindi $main::var è assolutamente identico a $main::main::var e $main::main::main::var. Questa tecnica di autoinclusione ha permesso di scrivere il Perl Debugger, che è un normale modulo Perl il quale, quando caricato insieme al vostro programma, consente il debugging interattivo.

3. Il concetto di Modulo

Un modulo è l'unità minima di riutilizzabilità di codice in Perl. Il nome del file che contiene il Modulo è il nome stesso del modulo (con preservazione della capitalizzazione delle lettere), con il suffisso ".pm". Inoltre la gerarchizzazione dei Moduli si traduce in una gerarchizzazione delle directory che contiene i moduli. Il modulo Tools::Editors::Writer è rappresentata da Tools/Editors/Writer.pm. Se esiste un modulo Tools::Editors::Writer::Macros esisterà anche per conseguenza il file Tools/Editors/Writer/Macros.pm!

Vediamo prima come utilizzare un modulo.

Come usare un modulo — Fig. 5

```
1: #!/usr/bin/perl
2:
3: use Cwd;
4:
5: my $cwd = getcwd();
```

oppure

Come richiedere un modulo — Fig. 6

```
1: #!/usr/bin/perl
2:
3: BEGIN { require Cwd; }
4:
5: my $cwd = Cwd->getcwd();
```

La differenza tra le due sintassi stà nel fatto che la prima richiede il modulo e ne importa tutti i simboli importabili nel namespace dal quale si stà chiamando il modulo. La seconda invece richiese esclusivamente il modulo senza importare i simboli che questo provvede. Per questo nel secondo esempio abbiamo dovuto usare l'operatore di dereferenziazione (->) per avere accesso alla sub getcwd().

4. Esempi di dichiarazioni di Moduli

Vediamo al volo un esempio di dichiarazione di un modulo:

Esempio di diachiarazione di un modulo — Fig. 7

```
 1: #!/usr/bin/perl
 2:
 3: package Tools::MyCwd;
 4: require Cwd;
 5: @ISA = qw/Cwd/;
 6: @EXPORT = qw/getcwd/;
 7: @EXPORT_OK = qw/getcwd oldcwd/;
 8:
 9: sub getcwd {
10:
11:   [ ... ]
12:
13: }
```

Come prima cosa dichiariamo il package del nostro Modulo. (Questo comporta che il nome del file, nel quale salveremo il modulo, sia Tools/MyCwd.pl). Subito dopo richiediamo il modulo Cwd. Questo ci serve perché il nostro package vuole estendere il package Cwd. Infatti questo modulo è stato introdotto nell'array @ISA, che contiene un elenco dei moduli all'interno dei quali cercare i simboli che non vengono reperiti nel package corrente. (Attenzione, stiamo un minimo sconfinando nella programmazione orientata agli oggetti, ma come vedremo non ci passa poi tanto!)

Successivamente definiamo gli array @EXPORT ed @EXPORT_OK. Questi due array contengono rispettivamente l'elenco dei simboli che vengono automaticamente esportati in caso di use, e l'elenco dei simboli che possono essere richiesti esplicitamente dal Modulo. Se un simbolo non figura qui e viene richiesto (es. use Tools::MyCwd qw/getcwd gettime/;), Perl segnalerà un errore!

Inizio Capitolo — Indice

Capitolo 8.
La programmazione orientata agli oggetti

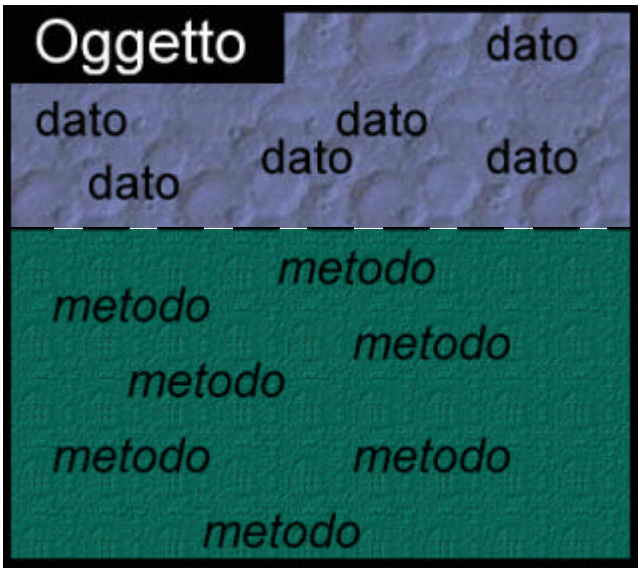
-

1. Cenni di programmazione orientata agli oggetti

Object Oriented Programming è una sigla che definisce un modo radicalmente diversa di concepire un programma. Nella programmazione strutturata canonica (ossia quella che abbiamo affrontato sino ad oggi), i dati e le procedure che li manipolano sono due oggetti distinti. Le strutture dati sono da un lato e spesso sono globali all'intero programma. Il codice che manipola questi dati è l'altra parte distinta, che può essere applicata a differenti strutture dati, purché caratterizzate dalla medesima "ossatura" (pena imprevedibili conseguenze sui dati stessi).

Nella Programmazione Orientata agli Oggetti invece la **Classe** è l'origine di tutta l'impostazione di un programma. La classe è il concetto che astrae la descrizione di una parte della struttura dati, combinata con la parte del programma che ne manipola, interroga ed imposta i contenuti.

Il reale vantaggio di questa concezione del programma è che il programmatore che usa un oggetto scritto da altri non tocca mai i dati con mano. Tutta la manipolazione dei dati è affidata alle subroutine che sono associate a quella particolare struttura dati.



Come potete notare l'oggetto contiene tanto la parte dati quanto la parte codice. Le subroutine che trattano con il dato, essendo strettamente correlati con QUELLA struttura dati, vengono chiamate *metodi* (in quanto costituiscono i metodi per modificare la struttura dati).

Per ogni classe esiste sempre almeno un metodo chiamato **costruttore** che serve a istanziare la classe, serve cioé a "costruire" un oggetto (ossia una copia della classe con però propria identità). Ad esempio, se abbiamo una classe Lampadina che descrive come sono fatte le lampadine (attenzione! TUTTE le lampadine di questo mondo, ossia quali parti hanno in comune), dalla classe (che di fatto è solo un "*modello*" di quell'oggetto) occorre ricavare un singolo oggetto al quale poter dare una specifica identità, ossia un set di valori che lo distinguano da tutti gli altri oggetti istanziati dalla sua classe.

Potremo quindi, attraverso il costruttore, ricavare una lampadina da 50W e una da 120W partendo dalla stessa classe. Per tradizione il costruttore in molti linguaggi si chiama new oppure si chiama come la classe stessa (nel nostro caso Lampadina). Perl non pone questo tipo di restrizioni e consente al programmatore di scegliere il nome che preferisce. Tuttavia in molti casi il programmatore chiama il costruttore new per una serie di motivi che capiremo in seguito.

Altra nota di distinzione di Perl da altri linguaggi di programmazione è che una classe può contenere più di un costruttore, che eventualmente agisca in maniera particolare a seconda dei dati forniti.

2. La Programmazione Orientata agli Oggetti in Perl

Non è molto quello che cé' da imparare per programmare Perl Object Oriented! Questo perché Perl usa concetti e strutture già note, ma con un paio di accorgimenti in più!

Primo: *Un oggetto è solo una "cosa referenziata" che sa da quale classe proviene.*

Secondo: *Una classe è solo un package che contiene i metodi per trattare con un particolare oggetto.*

Terzo: *Un metodo è solo una subroutine che riceve come primo parametro un reference alla classe della quale fa parte*

Lasciamo da parte la prima affermazione che si capisce meglio in seguito.... Come è costruito un oggetto in Perl? Vediamone uno che rappresenti una lampadina!

Un primo oggetto "Lampadina" Fig. 1

```
1: #!/usr/bin/perl
2:
3: package Lampadina;
4:
5: sub new {
6:   my $self = {};
7:   bless $self;
8:   return $self;
9: }
```

Alla linea 3 indichiamo che il package corrente è Lampadina. Questa operazione, in un contesto Object Oriented include anche l'implicita assunzione che il package che stiamo

creando è in reatà anche una classe, ossia l'astrazione di un oggetto.

Ora pensate al punto uno: Un oggetto è solo una "cosa referenziata" che sa da quale classe proviene. Guardate la linea 6. La variabile $self È il nostro oggetto! self è la "cosa referenziata" che sa da quale classe proviene. Prendiamocela comoda. Un oggetto, o almeno il suo midollo, è in realtà solo una variabile che subisce successivamente un trattamento speciale nel costruttore. Il motivo della scelta di una hash risulterà evidente in seguito. Quello che pero ora dovete pensare è che $self sia il nostro oggetto.

Bene! Studiamoci ora la riga 7, quella che "*fà la magia*"! bless è una parola chiave di Perl che "*oggettifica*" una variabile. Questo è il trattamento speciale di cui parlavamo prima. Con bless la variabile viene legata alla classe nella quale è definita. Infatti il costruttore new come ultima cosa ritorna la variabile.

3. Creiamo il primo oggetto

In realta' il nostro costruttore è ancora un po' scarno. Ma, anche se per ora fa poco senso, creiamo la nostra prima lampadina:

La prima "Lampadina" Fig. 2

```
1: #!/usr/bin/perl
2:
3: use Lampadina;
4:
5: my $lampadina = new Lampadina;
```

La direttiva use dice al compilatore di caricare la libreria (Modulo) Lampadina. Con la riga 5 assegnamo alla variabile $lampadina il reference ritornato dal costruttore new della nostra classe Lampadina.

Ora $lampadina *contiene* la nostra "*istanza*" di Lampadina. Attraverso il reference alla classe potremo in seguito chiamare i metodi della nostra lampadina.

4. Descriviamo una lampadina

Quali sono le caratteristiche di una lampadina? La potenza, il colore della luce, il tipo di attacco (Edison o a baionetta). Decidiamo di dare alla nostra classe la capacità di poter descrivere la lampadina secondo questi parametri. Per far questo dobbiamo decidere dove sistemare questi dati e con quali metodi modificarli.

Partiamo con i dati, in modo da capire perché di solito si usa una hash come *cuore* di un oggetto. Per ciascuno dei nostri dati definiamo una chiave nella hash. E per coseguenza partiamo dalla definizione dei metodi che su essi interagiscono. Una caratteristica fondamentale di una lampadina è se essa sia accesa o spenta. Per questo definiamo il metodo switch ed il metodo state per cambiare lo stato dell'accensione e interrogare la lampadina sul suo stato.

Miglioriamo la lampadina Fig. 3

```
 1: #!/usr/bin/perl
 2:
 3: package Lampadina;
 4:
 5: sub new {
 6:   my $self = shift() || {
 7:     stato => 'spenta',
 8:     potenza => '0W',
 9:     attacco => 'baionetta',
10:     luce => 'gialla',
11:   };
12:   bless $self;
13:   return $self;
14: }
15:
16: sub switch {
17:   my $self = shift;
18:   $self{state} = $self{state} eq 'accesa' ? 'spenta' : 'accesa';
19: }
20:
21: sub state {
22:   my $self = shift;
23:   return $self{state};
24: }
25:
26: package main;
27:
28: my $lampadina = new Lampadina({
29:   stato => 'spenta',
30:   potenza => '60W',
31:   attacco => 'edison',
32:   luce => 'bianca',
33: });
34:
35: if ( $lampadina->state() eq 'spenta' ) {
36:   $lampadina->switch();
37:   print "la lampadina ora è accesa\n";
38: } else {
39:   $lampadina->switch();
40:   print "la lampadina è spenta\n";
41: }
```

Quali le differenze? Molte direi!

Prima di tutto una piccola grande modifica al costruttore: il cuore del nostro oggetto, la hash $self viene ora scelta prima (se possibile) una reference di hash che viene passata come primo argomento del metodo new. In caso questa non sia disponibile viene assegnata una nuova hash contenente valori di default. Tuttavia non è il nostro caso dato che l'oggetto Lampadina da noi creato contiene una hash predefinita di valori che ne cambia ben 3 su 4 rispetto al default.

Poi abbiamo la prima definizione di metodi oltre al costruttore! Il primo, switch cambia alla linea 18 lo stato della lampadina, il secondo, state ritorna lo stato della lampadina. Entrambi hanno come prima linea di codice my $self = shift;. Ricordate il terzo punto? Un metodo è solo una subroutine che riceve come primo parametro un reference all'oggetto del quale fa parte! Semplice! Quindi all'interno del metodo dobbiamo recuperare questo reference in una variabile, dato che così abbiamo la possibilità di interagire con la struttura dati del nostro oggetto!

Per esempio il metodo switch cambia lo stato della lampadina (accesa/spenta) eseguendo una verifica su $self{stato}: se è "accesa" allora cambia in "spenta", in caso contrario cambia in "accesa". Notate come il nostro programma in seguito esegua una verifica su $lampadina->state() e non sulla chiave della hash (alla quale per altro non ha nemmeno accesso). Questo perche' il dato deve essere protetto dall'utenza che usa l'oggetto, presumendo che l'utenza possa anche non conoscere i possibili valori di $self{stato} e supponendo che possa quindi per errore impostare quella chiave a "senza_corrente" oppure a "rotta".

5. I metodi e l'operatore ->

Come risulta dall'esempio precedente, l'operatore -> (detto operatore di dereferenziazione) serve a chiamare un metodo di un oggetto. Ad esempio abbiamo usato la sintassi $lampadina->state() per chiamare il metodo state dell'oggetto $lampadina. La sintassi è in tutta analoga a state $lampadina, ma l'operartore -> effettua la dereferenziazione e passa al metodo come primo parametro il reference all'oggetto.

6. Ereditarietà

In Programmazione Orientata agli Oggetti è comune riferirsi ad una caratteristica di un oggetto: la sua **ereditarietà**. Con questo concetto si intende esprimere chi siano i "*genitori*" dell'oggetto, ossia da chi l'oggetto discenda. Dai propri genitori un oggetto eredita tutti i metodi che in esso non vengono direttamente ridefiniti.

In Perl l'ereditarietà viene implementata con un meccanismo estremamente semplice: l'array @ISA, lo stesso già usato dai Moduli. Quando un oggetto non definisce un metodo che da esso viene chiamato (es. $lampadina->svita()) Perl attiva un meccanismo di ricerca tale per cui estrae a turno dall'array @ISA tutti gli oggetti che sono genitori di quello corrente; per ciascuno di questi oggetti estratti cerca il metodo in esso *e in tutti gli oggetti definiti dall'array @ISA di questo* e così via, ricorsivamente. Osservate l'oggetto Faro definito di seguito.

Ereditarietà Fig. 4

```
 1: #!/usr/bin/perl
 2:
 3: package Faro;
 4:
 5: use Lampadina;
 6: @ISA = ( Lampadina );
 7:
 8: sub new {
 9:   my $self = {};
10:   bless $self;
11:   return $self;
12: }
13:
14: sub punta {
15:   my $self = shift();
16:   my $direction = shift();
17:   $self{direction} = $direction;
18: }
```

La classe Faro estende la classe Lampadina. Se ora creiamo un oggetto Faro, potremo:

Applicazione pratica di ereditarietà Fig. 5

```
1: #!/usr/bin/perl
2:
3: use Faro;
4:
5: my $faro = new Faro;
6: $faro->switch();
7: $faro->punta('Nord');
```

Notate come la classe Faro non contenga il metodo switch che viene invece definito dalla classe Lampadina. Il metodo viente comunque recuperato attraverso la gerarchia degli array @ISA (che in questo caso è composta da solo l'oggetto Lampadina). Né nella nostra applicazione si menziona esplicitamente la classe Lampadina. È richiesto il solo caricamento del Modulo che contiene Faro (use Faro;).

Inizio Capitolo Indice

Capitolo 9.
Funzioni notevoli

-

Diamo un occhio alle funzioni notevoli di Perl. Tenete presente nella lettura di questo capitolo che non si vuole dare una descrizione completa di TUTTE le possibili funzioni di Perl, bensì un'assaggio. Per ulteriori informazioni sui comandi trattati o per conoscere altri comandi si rimanda alla lettura della man page di Perl.

Ricordate che tutte le funzioni qui esposte lavorano su $_ o su @_ oppure @ARGV se non viene specificato uno scalare o un array sul quale operare.

1. chop e chomp

chop $var elimina dalla variabile specificata (di tipo scalare) l'ultimo carattere. chomp $var invece elimina dalla variabile l'ultimo carattere se questo è un carattere di separazione (spazio o tab) oppure se è un "a capo" (\n). È utile usare chomp sulle righe di testo lette da un file.

chomp Fig. 1

```
 1: #!/usr/bin/perl
 2:
 3: open IN, "/var/log/syslog";
 4: while (<IN>) {
 5:   chomp;
 6:
 7:   [ ... ]
 8:
 9: }
10: close IN;
```

2. push, pop, shift, unshift e splice

aggiunge elementi ad un array in coda, mentre li aggiunge in testa. toglie un elemento ad un array dalla coda e lo restituisce, shift fa lo stesso ma lavorando in testa.

push e unshift Fig. 2

```
1: #!/usr/bin/perl
2:
3: push @array, "elem1", "elem2", "elem3";
4: while ( @array ) {
5:   print shift @array, "\n";
6: }
```

dà come output

```
elem1
elem2
elem3
```

splice @array, OFFSET, LENGTH, LIST elimina (e restituisce) una porzione di un array (iniziante a OFFSET e lunga LENGTH) e la sostituisce con LIST (se fornita).

3. defined

defined *simbolo* ritorna 1 se il simbolo è definito. Si può usare su uno scalare, un array, una hash o una sua chiave, su una subroutine o un filehandle.

defined Fig. 3

```
1: #!/usr/bin/perl
2:
3: if ( defined $ENV{SHELL} ) {
4:   print "You are using $ENV{SHELL} as shell\n";
5: }
```

4. open e close -- da concludere

Il comando open apre un canale di comunicazione con un file (filehandle). Attraverso questo canale è possibile leggere e modificare in contenuti di un file, aprire sessioni di un programma e leggere il suo output oppure scrivere il suo input. close invece chiude un canale aperto. I canali lasciati aperti vengono chiusi automaticamente al termine dell'esecuzione anche se è buona pratica chiuderli sempre esplicitamente.

Questi canali sono chiamati *filehandle* e sono il primo argomento specificato a open. Proviamo subito a mettere le mani nel codice:

open su /etc/passwd — Fig. 4

```
1: #!/usr/bin/perl
2:
3: open PASSWD, "</etc/passwd";
4: while (<PASSWD>) {
5:   print;
6: }
7: close PASSWD;
```

Questo semplice script apre un filehandle di nome PASSWD sul file /etc/passwd; entra in un ciclo while tale per cui, finché ci sono contenuti nel file, questi vengono posti una riga alla volta in $_ e stampati attraverso print. Al termine di tutto, close chiude il filehandle.

Il primo argomento di open è il nome del filehandle che vogliamo aprire. Il secondo invece è il nome del file al quale vogliamo collegare a quel filehandle. Avrete notato che prima del nome del file abbiamo specificato un carattere aggiuntivo (<) che indica che il file deve essere aperto in sola lettura. Ne esistono altri che elenchiamo qui:

Possibili aperture di un file o di un processo — Tabella 1

Operatore	*Posizione*	*Significato*
<	Inizio	Apre il file in lettura
>	Inizio	Apre il file in scrittura
|	Inizio	Apre una pipe a un comando e ne scrive l'input
|	Fine	Apre una pipe a un comando e ne legge l'output

ATTENZIONE: non è possibile scrivere e leggere da un pipe contemporaneamente usando un filehandle aperto con open!!!

5. unlink, mkdir, chmod

Perl ha un nutrito numero di subroutine che emulano i corrispettivi comandi UNIX per cancellare file, creare directory, cambiare i permessi ad un file e così via.

Comandi per i file — Fig. 5

```
1: #!/usr/bin/perl
2:
3: unlink "/var/lock/daemon.lock";
4: mkdir "/var/lock/daemon/";
5: open OUT, ">/var/lock/daemon/lock";
6: print OUT 1;
```

```
7: close OUT;
8: chmod 0755 "/var/local/daemon/lock";
```

6. split e join

split divide una stringa arbitraria di testo in una lista, usando il primo parametro come separatore.

split Fig. 6

```
1: #!/usr/bin/perl
2:
3: my $string = "Corso di   Perl";
4: my @array = split / +/, $string;
```

Notate come il primo parametro sia stato messo fra slash! Questo perché in realtà è una **Regular Expression** (questa in particolare significa *uno o più caratteri di spazio*).

join al contrario unisce un array o una lista di elementi con una sequenza di caratteri (il primo parametro).

join Fig. 7

```
1: #!/usr/bin/perl
2:
3: my @array = ( 'uno', 'due', 'tre' );
4: my $string = join '', '', @array;
```

Attenzione: join è sufficientemente furbo da evitare di aggiungere il pattern di giunzione anche **dopo** l'ultimo elemento. Non serve quindi toglierlo con una regexpr o una serie di chop.

7. keys e values

La keyword keys consente di ottenere un array contenente tutte le chiavi di una hash. In questo modo è possibile esplorare il contenuto di una hash. values per contro restituisce un array contenente tutti i valori della hash specificata. Ricordate sempre che tanto le chiavi quanto i valori non hanno nessuna presunzione di ordinamento. Per ordinarli si può utilizzare il comando sort:

keys e sort — Fig. 8

```
 1: #!/usr/bin/perl
 2:
 3: %h = (
 4:   italy => 'pizza',
 5:   france => 'crepes',
 6:   germany => 'wurstel',
 7: );
 8:
 9: for my $key ( sort keys %h ) {
10:   print "$key is associated to " . $h{$k} . "\n";
11: }
```

8. delete

delete cancella una chiave da una hash.

delete — Fig. 9

```
1: #!/usr/bin/perl
2:
3: delete $ENV{SHELL};
```

9. die e exit

die ed exit terminano entrambe l'esecuzione del programma. La differenza fra le due risiede nel fatto che die consente di comunicare all'utente un messaggio testuale mentr exit ritorna solo un codice numerico di errore.

die ed exit — Fig. 10

```
 1: #!/usr/bin/perl
 2:
 3: print "Dammi il limite superiore della serie: ";
 4: my $int = <STDIN>;
 5: chomp $int;
 6: $int =~ /^\d+$/ or die "\$int non e' valido!\n";
 7:
 8: for $f ( 0 .. $int ) {
 9:   print $f, "\n";
10: }
11:
12: exit(0);
```

$int contiene il limite alto della serie che inizia da zero e viene stampata in sequenza. Se la variabile non contiene un numero il programma termina la sua esecuzione con l'uso di die. Dopo aver eseguito tutto il ciclo for il programma termina *esplicitamente* la sua esecuzione mediante una chiamata a exit alla quale passa un numero intero che costituisce il suo *exit status*, ovvero un codice che indica numericamente se il processo è terminato correttamente o, in caso contrario, quale errore abbia incontrato.

Inizio Capitolo Indice

Indice

Capitolo 10.

Le Regular Expressions

1. Cenni

Le Regular Expressions sono un sistema per definire pattern di ricerca che consentano l'individuazione e l'eventuale sostituzione di porzioni di testo.

Le regexpr seguono alcune basilari regole:

- Le regular expression si scrivono per convenzione comprese fra due slash '/'

- Le lettere ed i numeri non hanno altro significato se non il loro proprio naturale

La regexpr /a/ significa un singolo carattere "a" e null'altro...

- I caratteri possono essere raggruppati in classi

Una classe di caratteri è l'enumerazione dei singoli caratteri che la compongono racchiusa da parentesi quadre. Ad esempio /[abc]/ è la classe composta dai caratteri "a", "b" e "c". Questa regexpr descrive quindi le stringhe "a", "b", "c".

Esiste la possibilità di descrivere direttamente insiemi di caratteri: /[a-z]/ descrive tutte le lettere minuscole, /[0-9]/ tutti i numeri, /[B-D]/ i caratteri "B", "C" e "D".

Esiste anche la possibilità di negare una classe, cioé di descrivere tutti i caratteri che

NON fanno parte della classe dichiarata. La negazione si esegue inserendo un accento circonflesso ('^') all'inizio della regexpr, es: /[^abc]/ descrive **TUTTI** i caratteri escluse le lettere "a", "b" e "c".

- Si possono descrivere possibili stringhe alternative

La separazione delle alternative avviene con il carattere pipe ('|'), es: /pippo|pluto/. All'interno di una regexpr più complessa si può delimitare l'elenco delle possibili scelte con una coppia di parentesi tonde, come in: /asinto(to|ttico)/ che descrive le due parole "asintoto" e "asintottico".

- Un carattere, una classe o una scelta mutipla possono essere quantificati

... è possibile cioé comunicare quante volte uno di questi elementi può ripetersi. Il segno * indica che l'elemento si può ripetere *zero o più volte*, il simbolo + indica la ripetizione per una o più volte, il simbolo ? indica la ripetizione zero o una volta, ossia indica l'opzionalità di un elemento: può esistere o meno. Questi quantificatori devono essere giustapposti all'elemento al quale si riferiscono. Es: /times?/ descrive la parola "time" o la parola "times". Invece l'espressione /(Long time)?ago/ descrive le stringhe "Long time ago" e "ago".

- Il carattere backslash ("\") commenta il carattere successivo

... consente cioè al carattere successivo di perdere il suo significato particolare. Ad esempio, come è possibile specificare il caratter ? dentro una classe di caratteri? Semplice: /[\?;!]/. Una rudimentale classe di punteggiatura. Gli altri due caratteri non hanno un significato particolare, quindi non necessitano di backslash davanti. Questo non rigetta la possibilità che siano preceduti lo stesso da un backslash; semplicemente implica la ridondanza del backslash stesso.

Inizio Capitolo Indice

Capitolo 11.
Interfacce Grafiche

1. Interfacce Grafiche e Toolkits

Una *Interfaccia Grafica* è uno dei possibili sistemi con cui un utente può interagire con un programma. È basata su un sistema di oggetti chiamati *Widget* che raggruppati formano un *ToolKit*. Su piattaforme UNIX avete un'ampia scelta fra *Motif*, *Athena*, *FLTK*, *LibForm*, *OpenLook*, *Tk* e di recente *Gtk*.

Motif è il toolkit storico di UNIX (o almeno quello più usato da un pezzo a questa parte, prima si usavano le librerie Athena). Tk deriva da Tcl, un altro linguaggio interpretato. Gtk invece è il più recente; è nato per sviluppare The Gimp, uno splendido programma di manipolazione grafica, passato a Motif dopo che le limitazioni imposte da questo hanno cominciato a limitare lo sviluppo di Gimp.

Gtk è successivamente assurto a toolkit per lo sviluppo di un completo Desktop chiamato Gnome e di una numerosa serie di altri programmi che sono in parte poi stati portati per avvalersi delle funzionalità offerte da Gnome stesso.

Perl support nativamente Tk. Tuttavia non tratteremo questo toolkit in quanto è limitato in molte direzioni. In effetti è nato non per provvedere un complesso toolkit ma un toolkit veloce e piccolo adatto a piccole interfaccie. Ci occuperemo invece di Gtk.

Il primo motivo stà nel fatto che Perl sfrutta completamente Gtk, permettendo di avere accesso a tutti i suoi widget e con la possibilità di dialogare anche con Gnome.

Il secondo motivo è che avvalendosi di un *Interface Builder* chiamato glade risulta particolarmente rapida la stesura dell'interfaccia grafica. Il programma glade2perl si occupa di produrre il codice Perl che costruisce l'interfaccia per voi. A questo punto non vi resta che costruire il sistema di handler che gestiscono il verificarsi degli eventi.

Ah, già: cosa sono gli eventi?

2. Eventi

Un *Evento* è un qualsiasi tipo di azione che l'utente compie sull'interfaccia. Ad esempio la pressione di un pulsante È un evento. La selezione di un elemento in una lista di elementi È un evento. La discesa di un menù o la selezione di uno delle sue voci È un evento.

Con Gtk e glade risulta particolarmente semplice la scrittura di complessi programmi. Ho personalmente sviluppato in meno di un'ora l'intera interfaccia di un tool di gestione di account e gruppi per Server UNIX, comprensiva di dialog per l'inserimento di dati utente, modifica di un gruppo e quant'altro, senza escludere una simpatica finestra 'About'! ;-)

Ora pero' non esageriamo e partiamo dai fondamentali. Usiamo i widget di base di Gtk e cominciamo a fare qualcosa.

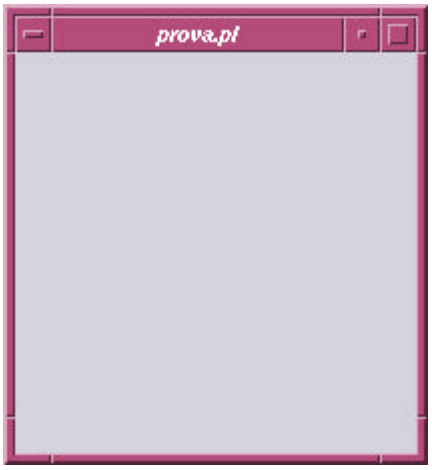
3. Una prima interfaccia

Costruiamo un semplice script che apra una finestra:

Solo una finestra — Fig. 1

```
 1: #!/usr/bin/perl
 2:
 3: #
 4: # Importiamo il modulo Gtk
 5: #
 6: use Gtk;
 7:
 8: #
 9: # Questa chiamata inizializza il modulo
10: #
11: init Gtk;
12:
13: #
14: # $win contiene il reference all'oggetto finestra
15: # appena creato
16: #
17: my $win = new Gtk::Window('toplevel');
18:
19: #
20: # Mostriamo la finestra
21: #
22: $win->show();
23:
24: #
25: # Facciamo partire il gestore (loop) degli eventi
26: # [ senza questa chiamata il programma aprirebbe la
27: # finestra e uscirebbe subito ]
28: #
29: main Gtk;
```

Ecco l'output del programma:



Come promesso apre una finestra (*e solo quella!!!!*) e si pone in attesa del verificarsi di qualche evento.

Aggiungiamo qualche giocattolo alla nostra finestra, ad esempio un bottone con una scritta che consenta di terminare il programma:

Aggiungiamo un pulsante — Fig. 2

```
 1: #!/usr/bin/perl
 2:
 3: use Gtk;
 4:
 5: init Gtk;
 6:
 7: my $win = new Gtk::Window('toplevel');
 8: $win->show();
 9:
10: #
11: # Queste quattro righe creano il bottone con laber 'Quit'
12: # collegano alla sua pressione (un evento, dunque) la chiamata
13: # alla sub anonima che chiude la finestra principale ed esce,
14: # lo posizionano nella finestra e chiedono alla finestra di
15: # mostrare tutti i suoi widget [ ->show_all() ];
16: #
17: my $button = new Gtk::Button('Quit');
18: $button->signal_connect('clicked', sub { $win->destroy(); exit; });
19: $win->add( $button );
20: $win->show_all();
21:
22: main Gtk;
```

Premendo il pulsante (che occupa tutta la finestra) viene prima chiusa la finestra e poi terminato il programma con un exit.

4. Containers

Per aggiungere un successivo bottone non sarà sufficiente chiamare nuovamente il metodo add() della finestra. Occorre aggiungere alla finestra principale un *container*. Si tratta di *meta widget* (ossia di widget che esistono, non si vedono direttamente ma se ne vedono gli effetti) il cui scopo è stabilire un ordine, un criterio con il quale posizionare i widget nella finestra o dentro altri widget.

I due più semplici (e che fanno proprio al caso nostro) sono i Box. Ne esistono due tipi: *verticale* e *orizzontale*. Proviamo ad usare un container verticale, contenente una entry ossia un riquadro che contenga testo, olte al pulsante per uscire:

Un bottone ed una entry — Fig. 3

```
 1: #!/usr/bin/perl
 2:
 3: use Gtk;
 4:
 5: init Gtk;
 6:
 7: my $win = new Gtk::Window('toplevel');
 8:
 9: #
10: # Creiamo il notro container verticale
11: #
12: my $vbox = new Gtk::VBox();
13: $win->add($vbox);
14:
15: #
16: # Creiamo la entry e la posizioniamo nel VBox
17: # [ i tre parametri che seguono sono
```

```
18: #   expand => puo' espandere oltre i suoi limiti?
19: #   fill => se espande puo' riempire lo spazio?
20: #   padding => quanti pixel ha intorno?
21: # ]
22: #
23: my $entry = new Gtk::Entry();
24: $vbox->pack_start( $entry, 0,0,0 );
25: $entry->set_text('Questo widget contiene testo');
26:
27: my $button = new Gtk::Button('Quit');
28: $button->signal_connect('clicked', sub { $win->destroy(); exit; });
29:
30: $vbox->pack_start( $button, 0,0,0 );
31: $win->show_all();
32:
33: main Gtk;
```

5. Un ultimo esempio

I commenti al codice dovrebbero essere sufficienti a questo punto per capire cosa faccia il programma.

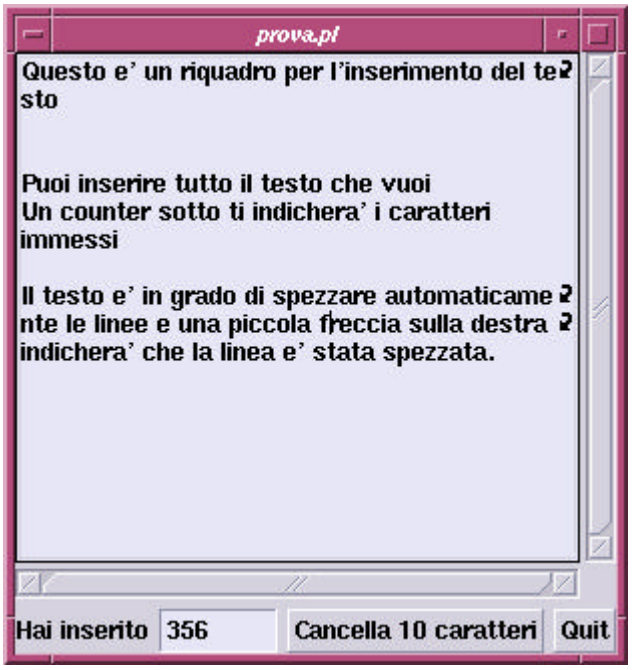
Una finestra un po' piu' complessa — Fig. 4

```
1: #!/usr/bin/perl
2:
3: #
4: # Include il supporto per Gtk
5: #
6: use Gtk;
7:
8: #
9: # Inizializza Gtk
10: #
11: init Gtk;
12:
```

```
13: #
14: # Crea la finestra e la dimensiona
15: #
16: my $win = new Gtk::Window('toplevel');
17: $win->height(300);
18: $win->width(300);
19:
20: #
21: # Il box che sviluppa in verticale
22: #
23: my $vbox = new Gtk::VBox();
24: $win->add($vbox);
25:
26: #
27: # La finestra scrollante che contiene
28: # il riquadro del testo
29: #
30: my $sw = new Gtk::ScrolledWindow();
31: $vbox->pack_start( $sw, 1,1,0 );
32:
33: #
34: # Il counter dei caratteri
35: #
36: my $cchar = 0;
37:
38: #
39: # Crea il riquadro per il testo
40: #
41: my $text = new Gtk::Text();
42: $sw->add( $text );
43: $text-
>insert(undef,undef,undef,"Questo e' un riquadro per l'inserimento del testo");
44: $text->set_editable(1); # Accetta input nel riquadro del testo
45:
46: #
47: # Il box orizzontale che contiene i bottoni
```

```
48: #a piede di pagina
49: #
50: my $hbox = new Gtk::HBox();
51: $vbox->pack_start( $hbox, 0,0,0 );
52:
53: #
54: #Una lable esplicativa
55: #
56: my $label = new Gtk::Label('Hai inserito');
57: $hbox->pack_start($label, 0,0,0 );
58:
59: #
60: #La entry del testo che conta i caratteri inseriti
61: #
62: my $charcount = new Gtk::Entry();
63: $hbox->pack_start( $charcount, 1,1,0 );
64:
65: #
66: #I due bottoni...
67: #
68: my $quit = new Gtk::Button('Quit');
69: $quit->signal_connect('clicked', sub { $win->destroy(); exit; });
70:
71: my $clear = new Gtk::Button('Cancella 10 caratteri');
72: $clear->signal_connect('clicked', sub { $text->set_point( 0 ); $text-
>forward_delete(10); });
73:
74: #
75: #Include i due bottoni
76: #
77: $hbox->pack_end( $quit, 0,0,0 );
78: $hbox->pack_end( $clear, 0,0,0 );
79:
80: #
81: #Chiede alla finestra principale di mostrare
82: #tutti i widget in essa contenuti
83: #
84: $win->show_all();
85:
86: #
87: #Collega un handler all'inserimento del testo
88: #
89: $text->signal_connect('changed', sub { $cchar++; $charcount-
>set_text($cchar); } );
90:
91: #
92: #Attiva il gestore degli eventi
93: #
94: main Gtk;
```

Eccone l'output:



Inizio Capitolo

Indice