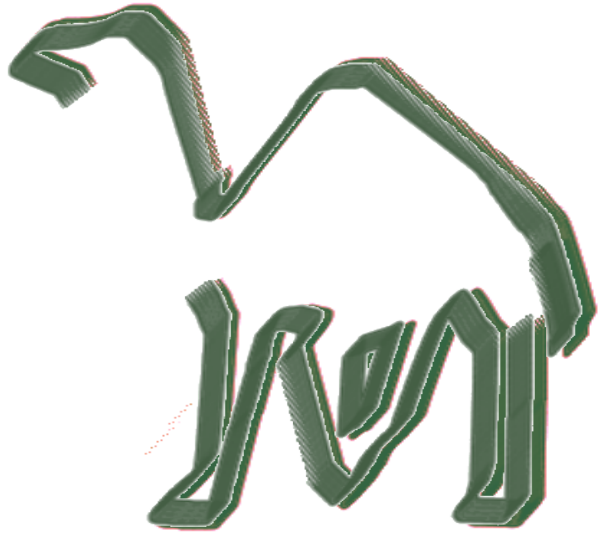
Programación y automatización. Recursos del administrador de sitios



Hugo Maza M.

Manual de Perl básico desde cero

Con ejemplos sencillos para automatización de procesos y tareas en servidores

Edición 0.1

Licencia de uso

Índice

Dedicatoria y Agradecimientos

Un poco de historia anticlínica...

Todo esto comenzó con una iniciativa intra-grupal de enseñanza, la cual constaba en que cada miembro del equipo de trabajo capacitara a los demás en lo que era bueno en su área.

Por razones algo obvias, fui yo el primero de la lista para compartir un poco mis escasos conocimientos de Perl con mis compañeros; conocimientos que me habían llevado a tener un reconocimiento entre los mismos por los resultados de mi trabajo.

La idea nace de crear un manual pequeño para apoyo del curso, sin embargo con el tiempo le fui agregando mas cosas, mas detalles, tareas y, repentinamente me di cuenta de que el material no era imprimible en la empresa, porque ya no eran unas cuantas hojas.

Esto condujo a no tomar en cuenta la impresión y a tomar en cuenta algunos comentarios de hacerlo mas legible en una PC.

De esta forma, me vi motivado para colocarle color al código, de la misma forma en que yo lo veía en mi editor de la laptop.

En un momento, las frágiles notas para el curso, se tornaron en un manual formal en color, que era mas fácil de asimilar para la gente que no estaba acostumbrada a leer código.

Así, este manual está considerado para aquellas personas que han tenido muy escaso contacto con el mundo de la programación (o nulo) y que desean llegar a crear procesos de automatización de tareas o de análisis de información, al terminar de leer el presente.

Es también útil como manual de bolsillo para recordar lo que en ocasiones se nos olvida sin razón alguna, es como decimos en México cuando jóvenes, un "acordeón".

Así, pongo a la disposición de todos los que desean ingresar al interesante mundo de la programación de Perl, este diminuto manual que espero sea muy útil como el primer paso en el aprendizaje y uso formal de Perl. Así mismo, para los instructores de nivel básico de Perl.

He de reconocer que sin la insistencia y el apoyo de mis ex-compañeros de trabajo en ese entonces de la empresa donde presto mis servicios, no hubiera podido realizar este pequeño pero interesante librillo de Perl.

Mis agradecimientos a ellos por el interés en aprender sobre Perl.

Igualmente manifiesto en gran medida mis agradecimientos a todas aquellas personas que de alguna u otra forma se interesaron por mi esfuerzo en el desarrollo del presente documento y que me sirvieron para seguir adelante con él.

Y muy especialmente a mi mujer Maye que soportó alejarme de las tareas domésticas algunos fines de semana.

A todos ellos y a usted que lo está leyendo... Gracias.

CAPÍTULO 1
Introducción

1. Introducción

1.1 ¿Qué es Perl?

Perl es un lenguaje de programación diseñado por Larry Wall creado en 1987. Perl toma características del C, del lenguaje interpretado shell (sh), AWK, sed, Lisp y, en un grado inferior, de muchos otros lenguajes de programación.

Estructuralmente, Perl está basado en un estilo de bloques como los del C o AWK, y fue ampliamente adoptado por su destreza en el procesado de texto y no tener ninguna de las limitaciones de los otros lenguajes de script.

Larry Wall comenzó a trabajar en Perl en 1987 mientras trabajaba como programador en Unisys y anunció la versión 1.0 en el grupo de noticias comp.sources.misc el 18 de diciembre de ese mismo año.

Perl se llamó originalmente "Pearl", por la Parábola de la Perla. Larry Wall quería darle al lenguaje un nombre corto con connotaciones positivas; asegura que miró (y rechazó) todas las combinaciones de tres y cuatro letras del diccionario. También consideró nombrarlo como su esposa Gloria. Wall descubrió antes del lanzamiento oficial que ya existía un lenguaje de programación llamado PEARL y cambió la ortografía del nombre.

El nombre normalmente comienza con mayúscula (Perl) cuando se refiere al lenguaje y con minúsculas (perl) cuando se refiere al propio programa intérprete debido a que los sistemas de ficheros Unix distinguen mayúsculas y minúsculas. Antes del lanzamiento de la primera edición de Programming Perl era común referirse al lenguaje como perl; Randal L. Schwartz, sin embargo, forzó el nombre en mayúscula en el libro para que destacara mejor cuando fuera impreso. La distinción fue subsiguientemente adoptada por la comunidad.

El nombre es descrito ocasionalmente como "PERL" (por Practical Extraction and Report Language - Lenguaje Práctico para la Extracción e Informe). Aunque esta expansión ha prevalecido en muchos manuales actuales, incluyendo la página de manual de Perl, es un retroacrónimo y oficialmente el nombre no quiere decir nada. La ortografía de PERL en mayúsculas es por eso usada como jerga para detectar a individuos ajenos a la comunidad. Sin embargo, se han sugerido varios retroacrónimos, incluyendo el cómico Pathologically Eclectic Rubbish Lister (Contabilizador de Basura Patológicamente Ecléctico).

1.2 El dromedario

Perl se simboliza generalmente por un dromedario (camello arábigo), que fue la imagen elegida por el editor O'Reilly para la cubierta de Programming Perl, que por consiguiente adquirió el nombre de El Libro del Dromedario. O'Reilly es propietario de este símbolo como marca registrada, pero dice que usa sus derechos legales sólo para proteger la "integridad e impacto de este símbolo". O'Reilly permite el uso no comercial del símbolo, y ofrece logos Programming Republic of Perl y botones Powered by Perl.

1.3 Características

La página de manual de Unix “perlintro” dice:

> *Perl es un lenguaje de propósito general originalmente desarrollado para la manipulación de texto y que ahora es utilizado para un amplio rango de tareas incluyendo administración de sistemas, desarrollo web, programación en red, desarrollo de GUI y más.*
>
> *Se previó que fuera práctico (facilidad de uso, eficiente, completo) en lugar de hermoso (pequeño, elegante, mínimo). Sus principales características son que es fácil de usar, soporta tanto la programación estructurada como la programación orientada a objetos y la programación funcional, tiene incorporado un poderoso sistema de procesamiento de texto y una enorme colección de módulos disponibles.*

El diseño de Perl puede ser entendido como una respuesta a tres amplias tendencias de la industria informática:

1. Rebaja de los costes en el hardware
2. Aumento de los costes laborales
3. Mejoras en la tecnología de compiladores.

Hay un amplio sentido de lo práctico, tanto en el lenguaje Perl como en la comunidad y la cultura que lo rodean. El prefacio de Programming Perl comienza con, "Perl es un lenguaje para tener tu trabajo terminado". Una consecuencia de esto es que Perl no es un lenguaje ordenado. Incluye características si la gente las usa, tolera excepciones a las reglas y emplea la heurística para resolver ambigüedades sintácticas. Debido a la naturaleza

indulgente del compilador, a veces los errores pueden ser difíciles de encontrar. Hablando del variado comportamiento de las funciones internas en los contextos de lista y escalar, la página de manual de perlfunc(1) dice "En general, hacen lo que tu quieras, siempre que quieras la coherencia."

Perl está implementado como un intérprete, escrito en C, junto con una gran colección de módulos, escritos en Perl y C. La distribución fuente tiene, en 2005, 12 MB cuando se empaqueta y comprime en un fichero tar. El intérprete tiene 150.000 líneas de código C y se compila en un ejecutable de 1 MB en las arquitecturas de hardware más típicas. De forma alternativa, el intérprete puede ser compilado como una biblioteca y ser embebida en otros programas. Hay cerca de 500 módulos en la distribución, sumando 200.000 líneas de Perl y unas 350.000 líneas adicionales de código C. Mucho del código C en los módulos consiste en tablas de codificación de caracteres.

(del punto 1.1 al 1.3, información extraída de wikipedia)

1.4 Cosas importantes para comenzar

Perl es un lenguaje interpretado, es decir, para ejecutar un script de Perl debemos indicarle al sistema operativo el intérprete que queremos utilizar para ejecutarlo.

Hay dos maneras de hacerlo:

a) Ejecutando el binario de Perl y pasarle como argumento el nombre completo del archivo de Perl. Ejemplo (en línea de comandos de Unix):

```
user@host ~$ perl script.pl
```

b) Insertando en la primera línea del archivo del script, los caracteres clásicos de inicio de algún script que reconoce el sistema operativo, el caracter de sharp y el signo de admiración (**#!**), seguido de la ruta del binario de Perl. Algo como esto:

```
#!/usr/bin/perl
....y comienza el script
```

En el segundo caso deberemos tener el archivo con permisos de ejecución, es decir, 755 ó -rwxr-xr-x.
Esto aplica para servidores con sistemas operativos basados en el estandard POSIX, ya que en algunos sistemas operativos que no tienen control de permisos de archivos, como Windows©, no es necesario buscar la forma de hacerlo ejecutable, ya que ahí todos los archivos lo son.

1.5 Cosas útiles al comenzar

Perl viene por defecto instalado en todas (desconozco si alguna no) las versiones de Unix, y los sistemas operativos Unix-Like como GNU/Linux.
Y existen versiones instalables para casi todos los sistemas operativos.

Si sabes que tienes instalado Perl, hay manera de saber algo sobre él en línea de comandos:

perl -v : Muestra la versión del intérprete de Perl que estamos utilizando.
perl -V : Muestra información sobre la configuración del intérprete de perl.

Otras “curiosidades” de línea de comando:

perl -d *script* : Ejecuta el script bajo el depurador.
perl -w *script* : Da avisos sobre las construcciones con errores.
perl -x *script* : Empieza a interpretar el archivo que contiene el script cuando encuentra la referencia al intérprete, por ejemplo: #!/usr/bin/perl.

CAPÍTULO 2
El script de Perl

2. El script de Perl

Un script en Perl es una sucesión de instrucciones.

Cada instrucción sola debe terminar siempre con una punto y coma (**;**).

Cuando deseamos insertar comentarios en los scripts, estos deben ir precedidos por el símbolo de sharp (**#**). Todos los caracteres que siguen a este símbolo y que están contenidos en la misma línea se consideran comentarios y se ignoran para la ejecución del programa.

Como convención, se usan terminaciones para los scripts de Perl características, como por ejemplo: ***.pl*** ó ***.cgi*** . Aunque se puede usar casi cualquier terminación que deseemos o incluso ninguna, cuando sea necesario. Para ser ordenados, usaremos esta convención estandard.

2.1 Nuestro primer script de Perl

Universalmente famoso se ha hecho la impresión del “Hola mundo” al mostrar el primer script de cualquier lenguaje, que no lo podemos dejar de lado en esta humilde obra.

Abrimos nuestro editor de texto preferido en el equipo de prácticas y empezamos a escribir el código en Perl, el cual es mas sencillo de lo que parece.

Al respecto podemos recomendar el uso de algunos programas potentes para escribir código:
En GNU/Linux es muy bueno el programa Geany (http://www.geany.org/)el cual es sumamente amigable para remarcar la sintaxis de muchos lenguajes, o bien Bluefish de Open Office (http://bluefish.openoffice.nl/).
Si bien Geany es mas poderoso en el control de la sintaxis, Bluefish posee un cliente de FTP para “subir” al servidor los archivos recién modificados sin usar otro programa como Filezilla (http://filezilla-project.org/) por ejemplo.

En Windows© puedes usar Notpad++ (http://notepad-plus.sourceforge.net/es/site.htm). También existe Filezilla versión para este sistema operativo.

Comenzamos como habíamos comentado, con la información para el sistema operativo, seguida de nuestra primera experiencia con Perl, la impresión de una cadena:

```
#!/usr/bin/perl
print "Hola mundo! \n";
# Este es un comentario
```

Puntos importantes que aprender de este primer script:

En este caso vemos claramente que podemos “externar” la información a la pantalla mediante la función ***print***, la cual redirige la salida a un *manejador* especial que en este caso si no especificamos uno, es la salida estandard o terminal; la pantalla de nuestro monitor.
Posteriormente veremos este tema poco a poco.

Podemos ver también que la información que queremos imprimir está colocada entre comillas dobles.

El salto de línea, lo vemos comúnmente reflejado como “**n**”, diagonal inversa y la letra n.
Y es un caracter mas, pero es un caracter especial.
Así, podemos posicionar el cursor en otro renglón después de terminar de ejecutar el script.

Durante el transcurso del libro usaremos esta notación para algunos caracteres especiales como es el caso del usado en este primer script, salto de línea.
Mas adelante entenderemos de que se trata.

Una vez que hayamos verificado los permisos de dicho script estaremos listos para ejecutarlo simplemente tecleando en la línea de comandos:

```
user@host ~$ ./hola.pl
```

O bien:

```
user@host ~$ perl hola.pl
```

CAPÍTULO 3
Representación de los diferentes tipos de datos

3. Representación de los diferentes tipos de datos

El lenguaje Perl posee tres tipos de representaciones de datos:

1. **Escalares**.
2. **Arrays** o listas indexadas.
3. **Hashes** o arrays de indexación literal, o listas de asociación.

De ahora en adelante, llamaremos **variable** a toda representación de datos.

Los datos representados dentro de las **variables escalares** son las **cadenas**.

Los datos contenidos en los **arrays** o en los **hashes**, generalmente pueden contener variables escalares.

Estas tres representaciones permiten asociar a cada variable utilizada un tipo de cadena.

Es muy importante comentar que en Perl, ni las variables ni los tipos de datos dentro de las variables tienen que declararse antes de iniciar la construcción del programa.

> *Las variables se pueden declarar de manera dinámica conforme va transcurriendo el programa y poseen un valor predeterminado en función del contexto.*

O sea, que las podemos declarar en cualquier momento en cuanto las necesitemos.

Esto nos da una libertad completa con el uso de variables en comparación con otros lenguajes de programación como por ejemplo java©.

3.1 Los Escalares

El escalar representa el tipo de datos básico en Perl. Y nos permite representar números reales, números enteros y cadenas de caracteres.

Perl permite representar los datos con los que trabajamos de una manera muy fácil y directa.

Una cadena simple es representada, como en otros lenguajes, con un símbolo de pesos (**$**) seguida del nombre de la cadena, la cual admite en su nombre los caracteres simples

alfanuméricos y el carácter underscore (_) o guión bajo, y no debe comenzar con número.

Ejemplos de nombres de variables escalares que podemos usar:

```
$nombre_usuario;
$nombreUsuario;
$salida_calculo;
$resultadoTotal;
$resTotal;
$subTotal1;
$subTotal_2;          # caracteres alfanuméricos y underscore
```

3.1.1 Números reales y enteros

Los valores numéricos expresados literalmente se presentan en forma de valores reales codificados en doble precisión. Así pues, Perl no distingue entre enteros y flotantes.

Este formato interno se utiliza para todas las operaciones aritméticas.

Y aunque en otros lenguajes de programación, números y cadenas de caracteres son dos cosas muy distintas, en Perl ambos son escalares.

Así, es posible declarar una cadena numérica ingresándola dentro de comillas dobles o sencillas y trabajar con ella como número, aunque sin poseer precisión real.

Ejemplos de datos numéricos:

```
$x = 0.789;          # es un real
$y = 1.22e-22;       # es un real
$n = 765;            # es un entero
$i = -128;           # es un entero
```

Hay que hacer notar en estos ejemplos dos cosas:

Usamos el signo de pesos (**$**) para declarar la existencia de una variable.

Los datos o valores numéricos son asignados a las variables directamente sin usar comillas para el efecto, esto se debe a su valor numérico. Las cadenas, como veremos mas adelante, deben usar otros métodos para ser declarados.

Los valores **enteros** no pueden empezar con cero porque esto especifica un entero mediante su codificación octal. Valores octales comienzan con cero 0; los valores

hexadecimales comienzan con un 0x, y con 0b los binarios. Por ejemplo:

```
$x = 0377;          # equivale a 255 (representación octal)
$y = 0xff;          # equivale a 255 (representación hexadecimal)
$y = 0b11111111;    # equivale a 255 (representación binaria)
```

3.1.2 Cadenas de caracteres

Las cadenas de caracteres se especifican por medio de una sucesión de caracteres delimitada por comillas dobles (**"** .. **"**) o comillas simples (**'** .. **'**).

Estas dos representaciones son interpretadas por Perl en muy distinta forma.
Cuando van delimitadas por comillas dobles (**"** .. **"**), toda variable referida en el interior de la cadena se evalúa y es reemplazada por su valor. Esto se llama Interpolación de variables.

3.1.2.1 Interpolación de variables.

Veamos un ejemplo, en las instrucciones siguientes, la variable **$wld** es remplazada por la cadena de caracteres "mundo" en el interior de la variable **$str**.

```
$wld = "mundo";
$str = "¡Hola $wld!"; # se sustituye la variable $wld por su valor
print $str , "\n";    # imprime ¡Hola mundo!, además de un salto de línea
```

Por el contrario, en las cadenas de caracteres delimitadas por comillas simples, todos los caracteres permanecen intactos. Por ejemplo:

```
$wld = "mundo";
$str = '¡Hola $wld!'; # no se sustituye la variable $wld
print $str , "\n";    # imprime ¡Hola $wld!
```

La cadena **$wld** nunca es remplazada por el valor de esta variable. Imprime ***¡Hola $wld!***.

Así, si deseamos que una eminente sustitución de cadena (interpolación) **no** se lleve a cabo, debemos utilizar las comillas simples para crear el valor de la cadena sin la interpolación.

Esto aplica también cuando incluyamos caracteres especiales dentro de la cadena, muchos los veremos mas adelante.

Hay ocasiones que necesitamos usar comillas dobles para declarar una cadena por las variables internas que lleva dentro de ésta, pero (siempre hay un pero) también dentro de la misma cadena principal debemos usar comillas doble por alguna razón.

En este caso, podemos hacer uso del caracter especial backslash o diagonal inversa (\ ) seguido de las comillas dobles. La diagonal inversa, nos permite declarar el valor verdadero del siguiente caracter sin hacer uso de lo que pueda significar para Perl.

En este caso las comillas dobles significarían para Perl un final de cadena, pero si dentro de la cadena insertamos un (**\"**), Perl NO lo interpretará como final de cadena, sino como un caracter mas dentro de la cadena:

```
$miCadena = "Hola \"mundo\".\n";

print $miCadena;      # esto imprime: Hola "mundo". y un salto de línea
                      # para que el prompt del sistema operativo quede en
                      # una nueva línea
```

Note bien que las comillas con el backslash que las precede no significan el final de la cadena, sino que significan solo el caracter de comillas dobles.
Las comillas dobles después del salto de línea, si significan el final de la cadena.

En otro aspecto, en algunas circunstancias, es necesario “pegar” una cadena a unos caracteres locales para juntar ambos y que el resultado sea legible como la unión de estos.

En estos casos es necesario la intervención de una referencia al nombre de la cadena.
Pero mejor veamos un ejemplo:

```
$w = "w";
$var = "El sitio es $www.google.com \n";
print $var;    # ESTO IMPRIME: El sitio es .google.com
```

Esto imprime **El sitio es .google.com** ya que Perl reconoce una variable de nombre **$www**, la cual aún no está declarada y por lo tanto, no existe y no se imprime.
(Recordar siempre los caracteres admitidos en los nombres de cadena.)

Para poder hacer esta concatenación es necesario usar una referencia al nombre de la variable con los símbolos de llaves:

```
$w = "w";
$var = "El sitio es ${w}ww.google.com \n";
print $var;    # ESTO IMPRIME: El sitio es www.google.com
```

Esto hace que la cadena declarada **$w** cuyo contenido es **w** se concatene a los caracteres “ww” y forme una mezcla resultante de “www”, imprimiendose correctamente.

3.1.2.2 Concatenación de variables

Esto que acabamos de ver es un poco parecido a juntar dos variables para que al imprimir ambas, el resultado sea la UNIÓN de las dos variables.

Esto se llama concatenación de variables y el modo de hacerlo es el siguiente:

```
$hola = "Hola";
$wld = "mundo";
$holamundo = $hola . " " . $wld . "\n";
print $holamundo;     # imprime: Hola mundo
```

Lo explicamos:

La concatenación se lleva a cabo mediante los puntos, es decir, el punto concatena una o mas variables y/o cadenas, uniéndolas como si fueran una sola.

La variable $holamundo está formada por la unión de 4 cadenas, la cadena de la variable $hola; después se une otra que es una cadena de solo un espacio, luego la cadena mundo de la variable $wld y finalmente la cadena con el caracter del salto de línea...

\- - -

En otro aspecto, hay otro tipo de sintaxis que permite delimitar una cadena de caracteres para asignarlas a variables, son las etiquetas.

Se utilizan cuando la cadena de caracteres contiene varias líneas y/o comillas o apóstrofes. Su sintaxis es la siguiente:

```
$str = <<ETIQUETA;
....
....
ETIQUETA
```

donde la **ETIQUETA** es una cadena de caracteres cualquiera.
El fin de la cadena se determina por la nueva línea que contiene únicamente el identificador de la etiqueta.
Éste no debe ir precedido por un espacio ni marca de tabulación. Por ejemplo:

```
$msg = <<ETQT;
hola,
       buenos "días",
mundo,

ETQT
print $msg;
```

Al momento de correr el script se imprime lo siguiente:

```
user@host ~$ ./script.pl
hola,
       buenos "días",
mundo,
user@host ~$
```

Notamos que se imprime exactamente como declaramos la cadena.
Esto incluye los saltos de línea físicos, hasta el último de ellos, que nos sirve para posicionar el cursor del prompt del shell de Unix en una línea nueva.

Otra cosa que hay que notar es que la etiqueta de fin de cadena no lleva el punto y coma (**;**) característico de los finales de instrucción, porque el final de la instrucción ya fué declarado con el inicio de la etiqueta.

3.1.3 Boleanos

El tipo de cadena booleana existe al igual que en C, de modo implícito, es decir, un número es falso si es igual a cero y verdadero en el caso contrario.

Como el cero está asociado a la cadena vacía (""), cuando una cadena no contiene ningún caracter, equivale a tener un valor falso, o de cero.
Esto nos resulta útil cuando manejamos bifurcaciones, bucles y otras funciones de Perl.
Pero no se asuste, lo veremos un poco mas adelante a detalle.

Las variables en Perl se asignan de manera dinámica y se les asigna un valor predeterminado en función del contexto donde se trabaja.

> *En un contexto numérico el valor predeterminado de una variable es 0.*
> *En un contexto de cadena de caracteres, el valor predeterminado es la cadena vacía ("").*

Esto debe tomarse en forma muy seria ya que nos servirá para todo tipo de funciones

dentro de un desarrollo de script de Perl, así que grabémonos bien en la cabeza esta idea que acabamos de mencionar.

Por esto, es fácil darnos una idea de los valores que toman por defecto las variables cuando las declaramos.

En Perl, es muy común declarar una variable antes de usarse por ejemplo, en un bucle, y lo vemos con el siguiente ejemplo:

```
# declaramos la variable y no le asigamos ningún valor predefinido.
$variable;

# o bien:
$var = "";
```

En ambos casos la variable posee un valor de vacío, sin embargo, para efectos de validar con una variable si el proceso es verdadero o falso (como veremos mas adelante), ambas variables toman el valor de cero (**0**). Y cero es falso (verdadero en cualquier otro caso).

3.1.4 Contexto de uso de las variables

El tipo y el valor de las variables en Perl se determina a partir del contexto.
Así, una cadena de caracteres conteniendo un valor numérico será convertida en variable numérica en un contexto numérico, como ya lo habíamos mencionado.

Esto quiere decir que si declaramos una variable que represente una cadena de caracteres que sean solo números, si posteriormente la usamos dentro del contexto numérico, esta cadena se convierte a números.

Esto nos indica que podemos hacer operaciones algebráicas con estas variables como si se trataren de variables numéricas y no de cadenas de caracteres.

Ejemplo:

```
$pi = 3.1416;          # un número real
$var = "20";           # una cadena de caracteres
$suma = $pi + $var;    # sumamos ambas variables
print $suma , "\n";    # imprime 23.1416
```

En algún caso podremos tener una variable mezclada entre números y caracteres.

En este caso, **y solo si** la variable comienza con números, Perl solo tomará los números iniciales para calcular las operaciones aritméticas y el resto, desde el primer caracter alfabético que encuentre, serán ignorados:

```
$var_1 = "20cfv1";          # para Perl, vale 20 en un contexto numérico
$var_2 = 4;
$res = $var_1 / $var_2;     # Perl dice: 20 / 4
print $res , "\n";          # imprime 5
```

3.2 Los arrays o listas indexadas

Debemos antes que nada, comprender qué es una lista de datos.
Se expresa una lista de datos generalmente mediante el uso de paréntesis, y al declararlos, los separamos por comas (**,**):

```
(ElementoUno, ElementoDos, ElementoTres)
```

Algunas veces podremos prescindir de los paréntesis.

Si nos damos cuenta, hemos usado la función **print** por ejemplo, para mostrar en pantalla una lista de elementos:

```
print $var , "\n";
```

La mayoría de las funciones (como **print**) recorren iterativamente una lista de elementos para aplicar su función a cada uno de ellos.

La orden anterior podemos expresarla también de esta forma:

```
print ($res , "\n");
```

Dándonos cuenta de que la función **print** se aplica en realidad a una lista de elementos, y la recorre iterativamente para realizar su proceso sobre cada uno de ellos.

Ahora, cada elemento de la lista es una variable escalar a la que se le asocia un valor...

> *Un **array** es una lista de datos de tipo escalar, indexados numéricamente.*

Es una lista **ordenada** según los índices de sus elementos.

Para darle nombre a estas variables de tipo **array**, también deben usar caracteres

alfanuméricos y/o underscore, y deben siempre comenzar con letras.

Estas variables, de tipo array, se identifican por el prefijo arroba @. Por ejemplo:

```
@numeros = (2, 1, 667, 23, 2.2, 5, 6);
@letras = ("perro", "gato", "león");
@mezcla = ("hola mundo", 23, "adios", 31.234);
```

Cuando se trata de rangos de letras o números, es posible definir fácilmente ese rango de caracteres escalares mediante el uso de dos puntos seguidos, que unen los extremos del rango. Ejemplo:

```
# rango de letras minúsculas desde la "a" a la "z:
@alfabeto = ( a .. z );

# todos los números comprendidos entre "1" y "100" incluyendo estos:
@numeros = ( 1 .. 100 );

# mas rangos especiales
@array = ( 1.2 .. 5.2 );      # (1.2, 2.2, 3.2, 4.2, 5.2)

# cadenas con el operador espacial "qw"
@array = qw(Juan Pablo Pedro María Ana 45); # no necesitamos las comillas
                                # ni las comas para separar los elementos
```

En el último caso, vemos algo diferente a los anteriores, usamos “**qw**” para denotar que los elementos dentro de los paréntesis van entre comillas.

Este método funciona cuando ningún elemento posee un caracter de espacio, de lo contrario lo separaremos como varios elementos del array...

Ahora, si ya declaramos nuestra variable tipo array, y la tenemos bien definida dentro de nuestro script, se preguntará cómo leer o acceder a los elementos individuales dentro del array.

Bien, en este caso se usan los “**índices**” del array para indicar cual posición dentro de éste queremos accesar.

Todos los elementos tienen un índice, y los índices comienzan desde el cero (**0**).
Estos índices se colocan entre corchetes (**[ ]**) y la variable ahora la escribimos como un escalar, porque los elementos del array son escalares.

Así por ejemplo, en nuestro array mezclado de números y letras:

```
@mezcla = ("hola mundo", 23, "adios", 31.234);      # no podemos usar qw()
print $mezcla[0], "\n";       # imprime: hola mundo
```

Esto imprimirá desde luego, el primer elemento del array (el elemento cero)
En otros ejemplos con el mismo array:

```
print $mezcla[1], "\n";       # imprime: 23

print "$mezcla[0] - $mezcla[2]\n";          # imprime: hola mundo - adios
```

... y esto nos mostró mas detalles del uso de los índices de los arrays.

Igualmente podremos accesar varios elementos del array por medio de rangos:

```
@numeros = (2, 1, 667, 23, 2.2, 5, 6);
@nums = @numeros[1..3];       # @nums = (1, 667, 23)
```

Nuestro nuevo array @nums consta de los elementos del 1 al 3 del array @numeros, o sea, desde el segundo elemento hasta el cuarto (recordemos que comienzan desde el cero)...

Ahora, es posible asignar directamente los valores de los elementos dentro del array, directamente a variables, esto se logra mediante el uso de los paréntesis, que ya vimos que se usan para declarar la variable de tipo array o una lista ordenada:

```
@anims = ("perro", "gato", "león");  # Lo que existe de un lado...

($pet_1, $pet_2, $pet_3) = @anims;   # existe del otro

print $pet_1, "\n";           # esto imprime: perro
print $pet_2, "\n";           # esto imprime: gato
print $pet_3, "\n";           # esto imprime: león


# Y si solo queremos tomar el primer elemento:
@anims = ("perro", "gato", "león");

($pet_1) = @anims;

print $pet_1, "\n";           # esto imprime: perro


# Ahora vamos a "extraer" el primer elemento al array
($pet_1, @anims) = @anims;    # $pet_1 = perro
                              # @anims = ("gato", "leon")
```

En la última línea quitamos el primer elemento del array, quedandonos con una variable

escalar sola y un array de dos elementos.

Existe una función de Perl que hace esto último, sin embargo la veremos mas adelante.

```
# Y para regresarlo como estaba:
@anims = ($pet_1, @anims);    # @anims = ("perro", "gato", "león")
```

Así podemos jugar con las propiedades de los elementos del array para aprender un poco.

Finalmente, si vemos las siguientes líneas de código:

```
@a = (1,2,3);
@b = (5,6,7);
@c = (@a,4,@b,8);
```

Estas expresiones generan tres arrays, (1,2,3), (5,6,7) y (1,2,3,4,5,6,7,8). Esto también se llama ***interpolación***.

Cuando incluimos un array dentro de otro, Perl "mezcla" ordenadamante los array, insertando uno a uno todos sus elementos en la posición indicada de cada array.
Para hacer array de arrays deberemos aprender a crear arrays bidimensionales, que veremos posteriormente.

Para crear o inicializar un array vacío, hacemos esto:

```
@array = ();
```

Esto nos funciona también para “vaciar” una array existente, dejándolo sin elementos.

3.3 Los hashes, o arrays de indexación literal, o listas asociativas

Una lista asociativa está indexada por pares en lugar de índices (como el array). Se utiliza el signo de **%** para definir este tipo de lista asociativa.

Un elemento está indexado por el anterior, formando ambos, parejas del tipo (**llave**, **valor**), en donde llave = valor.

Accesamos el **valor** por medio de la **llave**.

Veamos el siguiente conjunto de elementos de un hash:

```
%cuotas = ("root", 1000, "maye", 5000, "hugo", 4000);
```

Si nos imaginamos un simil con los arrays, podremos definir que los elementos de un Hash, serían el elemento 0 con el 1, el elemento 2 con el 3 y el 4 con el 5.

En este Hash, root vale 1000, maye vale 256 y hugo vale 4000.

En este caso, para acceder al valor de cualquiera de los elementos de **%cuotas** debemos conocer su llave.

Al referirnos a los elementos de un hash usamos las llaves para encontrar el valor.

Por ejemplo:

```
%cuotas = ("root", 1000, "maye", 5000, "hugo", 4000);
$id = $cuotas{"maye"};                # $id = 5000
print $cuotas{"hugo"}, "\n";          # imprime 4000
```

Esta lista puede completarse añadiendo nuevos valores y asociando a cada llave, el valor correspondiente. Por ejemplo:

```
$cuotas{"alan"} = 350;
```

donde la llave es "alan" y el valor asociado es 350.

Es importante mencionar que en estas listas de asociación, las llaves no pueden repetirse, y es obvio puesto que si tratáramos de agregar un elemento al hash, y el cual no sabemos que ya existe, simplemente modificaremos el valor:

```
$cuotas{"maye"} = 800;
```

Aquí, como la llave “maye” ya existía, solo modificamos el valor, ahora la llave “maye” tiene un valor de 800 en lugar de 5000.

Ahora, este método simple de declarar pares de “**llave**-**valor**” puede prestarse a errores de dedo o teclado, sin embargo existe otra forma de declarar variables del tipo hash, y es mediante el uso de comodines como el “**=>**”.

Los pares se separan por comas.

Ejemplo:

```
%cuotas = ("root" => 1000, "maye" => 256, "hugo" => 4000);
```

O bien, en varias líneas para mejor visualización:

```
%cuotas = (
  "root" => 1000,
  "maye" => 256,
  "hugo" => 4000
);
```

De esta forma no nos “hacemos bolas” con las manos al momento de declarar un hash con una cantidad grande de pares llave – valor, porque organizamos la lista de pares, uno por cada línea a razón de **llave** => **valor**.

Hay que aclarar que tanto las llaves del hash como sus valores pueden ser cadenas de texto o números.

Así mismo se puede incluso declarar el valor de una llave, como una variable del tipo array. Esto nos lleva a crear variables multidimensionales.

3.4 Las referencias

Las referencias son un tipo de datos que permite hacer referencia a datos contenidos en otra entidad. No forman una nueva representación de datos, ya que éstos son tratados como un tipo más de escalar ($nombre).

Una referencia, para explicarlo mejor, es una variable, cuyo contenido actual es la dirección en memoria del contenido verdadero de la variable.
Esto es, que apunta hacia la dirección real en donde se guarda la variable en memoria.

Si usted conoce *C*, las referencias son el equivalente a los punteros en *C*.
A diferencia de *C*, Perl no deja "huella" de las referencias en memoria sino que tiene un mecanismo de "papelera de reciclaje" que consiste en un registro que posee las diversas referencias a cada elemento en memoria y las destruye cuando descubre que nadie hace referencia a él.

Para definir una referencia se usa el operador de referencia backslash, o diagonal invertida ("\").

No existe un operador de de-rreferencia, aunque sí los métodos.
Ejemplos prácticos de creación de referencias:

```
$refa = \$a;          # referencia a escalar
$refb = \@b;          # referencia a array
$refc = \%c;          # referencia a hash
$refx = \$refb;       # referencia a referencia
```

(Tambien existen referencias a funciones y referencias a objetos, pero no son objetivo de vista de este manual)

3.4.1 Array referenciado anónimo

Veamos otra forma de crear una referencia de **array**. Observe los corchetes (**[ ]**):

```
$arrayref = [ 'e1', 'e2', 'e3' ];
```

Aquí el array no tiene nombre. No creamos primero el array y después lo referenciamos con el backslash.

$arrayref es en realidad, una referencia a un arreglo anónimo.
$arrayref apunta a una dirección en memoria.

> *Cuando necesitamos crear un array referenciado anónimo, es necesario usar los corchetes (**[ ]**) para declarar las cadenas que contiene.*

```
$arrayRef = [ a, b, c, d, ...n ];
```

Y para accesar los elementos:

```
$array = [1, 2, 3, 4];
print $$array[2], "\n";       # imprime 3
```

Como se ve, el doble signo **$$** se usa para accesar un elemento de la referencia.

En Perl 5 se ha incluido un método de “abreviar” los elementos de las referencias:

```
$array = [1,2,3,4];
print $array->[2], "\n";      # imprime 3 (mas elegante)
```

Note los símbolos “**->**“ que se usan para aplicar una selección de la referencia.

$$array[n] y $array->[n] son lo mismo, ambos llaman a un elemento del array referenciado.

3.4.2 Hash referenciado anónimo

Ahora, otra forma de crear una referencia a hash. Observe las llaves " **{ }** ".

```
$hashref = { 'k1' => 'v1', 'k2' => 'v2' };
```

*Cuando necesitamos crear un hash referenciado anónimo, es necesario usar las llaves " **{ }** " para declarar los pares que contiene.*

```
$hashref = {
  'k1' => 'v1',
  'k2' => 'v2',
  'k3' => 'v3'
};
```

Aquí el hash no tiene nombre. No creamos primero el hash y después lo referenciamos con el backslash.

$hashref es una referencia a un hash anónimo.
$hashref apunta a una dirección en memoria.

Y para accesar los elementos del hash referenciado:

```
$array = {1 => 2, 3 => 4, 5 => 6};
print $$array{3}, "\n";       # imprime 4
```

Como se ve, el doble signo **$$** se usa para accesar un elemento de la referencia.
Igualmente, un método de "abreviar" los elementos de las referencias:

```
# el mismo array:
$array = {
  1 => 2,
  3 => 4,
  5 => 6
};
print $array->{3}, "\n";      # imprime 4
```

Note los símbolos "**->**" que se usan para aplicar una selección de la referencia.
$$array**{n} y **$array->{n} son lo mismo, ambos llaman a un elemento del hash referenciado.

Se pueden realizar referencias a referencias y arreglos a referencias de forma que los arrays multidimensionales se creen con mucha facilidad.

Cuando una referencia es de-referenciada se obtiene el dato real .

```
# generamos las referencias
$refa = \$a;          # referencia a escalar
$refb = \@b;          # referencia a arreglo
$refc = \%c;          # referencia a hash
$refx = \$refb;       # referencia a referencia

# extraemos las variables de las referencias
${$refa}              # de-referencia de $refa, esto es, el valor de $a
@{$refb}              # de-referencia de $refb, esto es, el valor de @b
%{$refc}              # de-referencia de $refc, esto es, el valor de %c
```

Si tratamos de imprimir una referencia tratándola como una variable, lo único que nos ha de mostrar es algo como esto:

```
print $ref, "\n";

SCALAR(0x8060284)
```

o quizás

```
HASH(0x8060308)
```

ó

```
ARRAY(0x9d5e880)
```

Que son las direcciones en memoria en donde realmente se almacena el contenido de la variable.
Y claro que nos da una idea del tipo de variable que contiene.

3.5 Arrays bidimensionales

Existirán muchas ocasiones en que será necesario crear matrices de datos en donde éstos estén ordenados (y sean accesados), en forma bidimensional.

Y podemos de manera muy fácil, crear arrays de este tipo de la siguiente manera:

```
@arraybidimensional = (
  ['Casa', 'Carro', 'Moto'],
  ['Perro', 'Gato', 'Caballo'],
  ['Calle', 'Ciudad', 'Pais'],
  [22, 303, 405]
);
```

Observe que, cada línea (lista) de datos está declarada mediante el uso de corchetes **[]**.

Y los elementos dentro de los corchetes es una array de elementos escalares.
Esto nos da una idea, que podemos declarar un array bidimensional a partir de varios arrays:

```
@bienes = ('Casa', 'Carro', 'Moto');
@animales = ('Perro', 'Gato', 'Caballo');
@direccion = ('Calle', 'Ciudad', 'Pais');
@numeros = (22, 303, 405);

@arraybidimensional = (
  [@bienes],
  [@animales],
  [@direccion],
  [@numeros],
);
```

Y tenemos exactamente el mismo array del principio.

Ahora, qué hacemos para accesar a cada elemento del array bidimensional ?
Bueno, ya sabemos que podemos accesar a un elemento simple de un array mono-dimensional con los corchetes, ahora solo agregamos un segundo corchete para acercarnos al dato escalar real de la matriz:

```
@array = (
  ['Casa', 'Carro', 'Moto'],
  ['Perro', 'Gato', 'Caballo'],
  ['Calle', 'Ciudad', 'Pais'],
  [22, 303, 405]
);

print $array[1][1] , "\n";          # imprime: Gato
                                    # o sea, el segundo elemento
                                    # de la segunda lista

print $array[0][0] , "\n";          # imprime: Casa
```

El primer corchete refiere a las listas, el segundo corchete refiere al elemento de esa lista en particular.

3.6 Arrays bidimensionales referenciados anónimos

Recordemos un poco, como crear un array referenciado anónimo:

```
$arrayref = ['xxx', 'yyy', 'zzz'];
```

Y accesamos a los datos de esta forma:

```
print $$arrayref[0];
```

o bien, de la manera mas común y elegante:

```
print $arrayref->[1];
```

Ahora, en igual forma como creamos un array bidimensional, podemos crear un array bidimensional referenciado anónimo:

```
$arrayref = [
  ['Casa', 'Carro', 'Moto'],
  ['Perro', 'Gato', 'Caballo'],
  ['Calle', 'Ciudad', 'Pais'],
  [22, 303, 405]
];
```

Nuevamente recalcamos que una variable referenciada es solo un dato escalar, por lo que es una variable que se declara con el signo de pesos **$**, así mismo, note que usamos los corchetes para declarar un array referenciado.

En este caso es un array referenciado bidimensional.
Y es anónimo porque nunca declaramos primero el array y después lo referenciamos con el backslash ($array = \@array), sino que lo creamos "al vuelo".

Bueno, ya tengo mi array bidimensional referenciado anónimo, y ahora, como lo acceso ? Simple:

```
print $$arrayref[0][0];      # imprime Casa
```

ó bien, de la manera mas común y elegante:

```
print $arrayref->[2][0];     # imprime Calle
```

El primer corchete refiere a las listas, el segundo corchete refiere al elemento de esa lista en particular.

3.7 Hashes multidimensionales

Aparte de los arrays bidimensionales, los hashes multidimensionales se utilizan para asociar mas de un valor a la clave, veamos el siguiente ejemplo :

```
%hash = (
        'Manzana'              =>      [4, "Delicious red", "mediano"],
        'Tocino Canadiense'    =>      [1, "paquete", "1/2 pound"],
        'Alcachofa'            =>      [3, "lata", "8.oz."],
        'Remolacha'            =>      [4, "lata", "8.oz."],
        'Sazonador Especies'   =>      [1, "bote", "3 oz."],
        'Tortilla'             =>      [1, "paquete", "16 oz."],
        'Salsa picante'        =>      [1, "bote", "8 oz."]
);
```

Para extraer un valor particular, al conjunto de valores debe tratársele como un array, ya que si nos damos cuenta, separados por coma son una lista de elementos.

```
print $hash{"Tocino Canadiense"}[1];        # imprime paquete
```

La instrucción anterior mostrará en pantalla el valor *paquete* que es el segundo elemento. Es posible asignar un array a un valor de clave.

```
@ajos = (4, "Ajo", "mediano");
$hash{"Ajo"} = [@ajos];       # tiene que ser dentro de corchetes
print $hash{"Ajo"}[1];        # imprime Ajo
```

Es posible asociar arreglos de longitud variable a una misma estructura hash. El siguiente programa permite imprimir un hash con arreglos de distinta longitud asociados a las claves.

```
foreach my $key (sort keys %hash) {
        print "$key: \n";
        foreach my $val ( @{$hash{$key}} ) {
                print "\t$val\n";
        }
                print "\n";
}
```

Posteriormente entenderemos este último bucle.

3.8 Hashes multidimensionales referenciados anónimos

También podremos crear hashes multidimensionales referenciados anónimos tan fácilmente como los arrays bidimensionales anónimos:

```
$hash = {
        'Manzana'              =>     [4, "Delicious red", "mediano"],
        'Tocino Canadiense'    =>     [1, "paquete", "1/2 pound"],
        'Alcachofa'            =>     [3, "lata", "8.oz."],
        'Remolacha'            =>     [4, "lata", "8.oz."],
        'Sazonador Especies'   =>     [1, "bote", "3 oz."],
        'Tortilla'             =>     [1, "paquete", "16 oz."],
        'Salsa picante'        =>     [1, "bote", "8 oz."]
};
```

Recordemos que para crear un hash referenciado usamos las llaves "{ }" para crearlos.

Y para accesar los valores de las llaves del hash:

```
print $$hash{"Tocino Canadiense"}[1] , "\n";
```

O bien, de la forma mas común y elegante:

```
print $hash->{"Tocino Canadiense"}[1] , "\n";      # imprime paquete
```

Esto nos debe imprimir "paquete", que es el segundo elemento del array que corresponde al valor de la llave "Tocino Canadiense".

- - -

CAPÍTULO 4
Operadores en Perl

4. Operadores en Perl

En Perl, se distinguen tres tipos de operadores de datos. Y dependen de la representación de datos sobre la que estamos actuando:

1. Operadores para escalares.
2. Operadores para arrays.
3. Operadores para hashes ó listas asociativas.

4.1 Operadores para escalares

Este tipo de operadores se pueden sub-dividir en:

- Operadores aritméticos.
- Operadores relacionales.
- Operadores lógicos o booleanos.
- Operador de selección.
- Operadores de asignación.
- Operadores de bits.

4.1.1 Operadores aritméticos

Adelantamos en un ejemplo anterior, que podemos sumar dos variables con el signo de **+** entre ambas variables, en el tema “Los Escalares” en 3.1.4.

Ahora mencionaremos algunos otros tipos básicos de operaciones entre variables escalares.

Resta:

(Nota: si va a realizar pruebas, no olvide la primera línea del archivo)

```
#!/usr/bin/perl
$var1 = 50;
$var2 = 40;
$resta = $var1 - $var2;
print $resta , "\n";          # esto imprime 10
```

Se usa el guión medio (o signo de resta) para denotar una resta.

División:

```
#!/usr/bin/perl
$var1 = 50;
$var2 = "2";
$division = $var1 / $var2;
print $division , "\n";       # esto imprime 25
```

Se usa el signo de slash, o barra inclinada para denotar una división.

Multiplicación:

```
#!/usr/bin/perl
$var1 = 50;
$var2 = 2;
$multi = $var1 * $var2;
print $multi , "\n";          # esto imprime 100
```

Se usa el asterisco para denotar una multiplicación.

Ahora, podemos hacer varias operaciones aritméticas dentro de una sola instrucción.

Existe una jerarquía de ejecución de operaciones aritméticas cuando hacemos varias de estas dentro de una instrucción.
Sin embargo, en forma práctica es mejor utilizar siempre los paréntesis para asegurarnos que la instrucción se ejecute en el orden que deseamos.

Ejemplo (con sustitución de variable):

```
#!/usr/bin/perl
$var1 = 50;
$var2 = 2;
$var3 = 20;
$var4 = 4;
$resultado = (($var1 * $var2) / $var3) - $var4;     # ((50 * 2) / 20) - 4
print "El resultado es $resultado \n";     # esto imprime: El resultado es 1
```

El mismo bus de variables, con diferente organización de paréntesis:

```
#!/usr/bin/perl

$var1 = 50;
$var2 = 2;
$var3 = "20";
$var4 = 4;
```

```
$resultado = ($var1 * $var2) / ($var3 - $var4);     # (50 * 2) / (20 - 4)

print $resultado , "\n";      # esto imprime: 6.25
```

Ahora, los operadores aritméticos se pueden usar en Perl para incrementar automáticamente el valor de una variable o bien, decrementarlos.

++ y **--** (dobles signos + y - , sin espacios)

Además de la acción de modificar la variable devuelven el valor de esta variable.

Los operadores de **incremento** o **decremento** pueden ir delante o detrás de la variable, teniendo diferente significado en cada caso.

Si el operador **++** se sitúa **después** de la variable se denomina de post-incremento, haciendo que primero devuelva el valor y después se incremente la variable.

Ejemplo:

```
$k = 2;
$n = $k++;            # el valor de k se asigna a n y después se incrementa k
print $n, "\n";       # imprime 2
print $k, "\n";       # imprime 3
```

Por otro lado, si el operador **++** se sitúa **antes** de la variable se denomina pre-incremento y hace que primero se incremente la variable y después devuelva el valor.

Ejemplo:

```
$k = 2;
$n = ++$k;            # primero se incrementa k y luego se asigna a n
print $n, "\n";       # imprime 3
print $k, "\n";       # imprime 3
```

Lo mismo sucede con el operador de doble signo de resta (--).

4.1.2 Operadores relacionales

En Perl, existen dos tipos de operadores relacionales:

Operadores de valores numéricos
Operadores de cadenas de caracteres.

La lista de operadores es equivalente.

Pero veamos la siguiente tabla:

Operador relacional	Numérico	Caracteres
Igual a	==	eq
No igual	!=	ne
Menor que	<	lt
Mayor que	>	gt
Menor o igual	<=	le
Mayor o igual	>=	ge

Perl evalúa las operaciones resultantes y devuelve falso o verdadero en su caso:

```
$dos = 2;
$tres = "tres";
($dos eq "dos")              # devuelve un valor falso o cero
($dos == 3)                  # devuelve un valor falso o cero
($dos == 2)                  # devuelve un valor verdadero o 1
($tres eq "tres")            # devuelve un valor verdadero o 1
```

Sobre el uso de estos valores devueltos no nos preocupemos ahora para que nos sirven, pronto lo veremos.

Aparte de los operadores que hay en la tabla cabe distinguir otros operadores característicos únicamente del lenguaje Perl.

cmp

Este operador es utilizado para comparar **caracteres**, de manera que, retorna 0 si los caracteres comparados son iguales, 1 si la cadena de la derecha se encuentra al comienzo de la de la izquierda, y -1 en el caso contrario. Para aclarar el funcionamiento de este operador he aquí un ejemplo:

```
'uno' cmp 'uno'       # devuelve  0
'uno dos' cmp 'uno'   # devuelve  1
'dos uno' cmp 'uno'   # devuelve -1
'dos' cmp 'uno'       # devuelve -1
```

<=>

Este operador se utiliza para comparar valores **numéricos**, retornando 0 cuando son iguales, 1 cuando el termino de la derecha es menor que el de la izquierda y -1 en el caso contrario.

```
1 <=> 1        # devuelve  0
12 <=> 1       # devuelve  1
1 <=> 21       # devuelve -1
```

4.1.3 Operadores lógicos o booleanos

Los operadores lógicos están ligados a los relacionales ya que normalmente estos operadores se usan entre resultados de operaciones relacionales.

and	**&&**
or	||
not	**!**

Estos operadores complementan los relacionales, evaluando dos o mas expresiones.

El operador lógico && evalúa dos expresiones o mas y devuelve verdadero solo en caso de que todas las expresiones evaluadas resulten verdaderas.

Por su parte, el operador || evalúa dos o mas expresiones y devuelve verdadero cuando solo una de ellas resulte verdadera, no importa si las demás resultan falsas.

Y una vez que se dispone de uno o varios datos de tipo booleano, estos se pueden combinar en expresiones lógicas mediante los operadores lógicos (AND, OR, NOT, ...). Un ejemplo de combinaciones de este tipo de expresiones serían:

verdadero AND falso --> falso
falso OR verdadero --> verdadero
verdadero OR verdadero --> verdadero
falso AND falso --> falso

Ejemplos prácticos:

```
$dos = 2;
$tres = 3;

($dos == 2 && $tres == 3)     # devuelve verdadero ó 1 porque ambas son
                              # verdaderas

($dos == 2 && $tres == 2)     # devuelve falso ó 0
                              # ya que una condición es falsa

($dos == 2 || $tres == 2)     # devuelve verdadero ó 1 ya que
                              # una de las dos expresiones es verdadera

($tres != 3)          # devuelve falso ó 0, ya que $tres sí es igual a 3
```

4.1.4 Operadores de selección

Es un operador triario que requiere una condición y dos expresiones que se ejecutan cuando los valores resultan verdadero o falso.

Perl ejecuta una expresión u otra dependiendo del resultado falso o verdadero de la condición.

Su formato es el siguiente:

> *Condición? Exp1: Exp2*
>
> *Perl evalúa la **Condición** y devuelve **Exp1** si es verdadero, si es falso, devuelve el valor de **Exp2**.*

Por ejemplo:

```
$x = 7;
$i = ( $x < 8 ? 6 : $x + 1 );# si $x<8 entonces i=6, si no $i=$x+1
print $i, "\n";              # imprime 6, porque $x es menor que 8
```

otro ejemplo:

```
$x = 7;
$i = ( $x < 5 ? 6 : $x + 1 );# si $x<8 entonces i=6, si no $i=$x+1
print $i, "\n";              # imprime 8 ( o sea $x + 1 )
                             # porque $x no es menor que 5
```

4.1.5 Operadores de asignación

Una asignación también es un operador que devuelve la variable modificada.

4.1.5.1 Operador de asignación =

Si no nos habíamos dado cuenta, ya hemos estado usando un operador de asignación en lo que llevamos leído, es el operador de asignación "**=**":

```
$x = 7;
```

Perl asigna a la vaiable $x el valor completo de lo que está del lado derecho de la expresión, mediante el operador de asignación **=**.

4.1.5.2 Operador de asignación +=

Incrementa el valor numérico de la variable del lado izquierdo con el valor de la cadena del lado derecho:

```
$x = 7;
$x += 2;       # ahora $x vale 9
```

4.1.5.3 Operador de asignación -=

Decrementa el valor numérico de la variable del lado izquierdo con el valor de la cadena del lado derecho:

```
$x = 7;
$x -= 2;       # ahora $x vale 5
```

*4.1.5.4 Operador de asignación *=*

Multiplica el valor numérico de la variable del lado izquierdo con el valor de la cadena del lado derecho:

```
$x = 7;
$x *= 2;       # ahora $x vale 14
```

4.1.5.5 Operador de asignación /=

Divide el valor numérico de la variable del lado izquierdo con el valor de la cadena del lado derecho:

```
$x = 8;
$x /= 2;       # ahora $x vale 4
```

4.1.5.5 Operador de asignación .=

Concatena la variable del lado izquierdo con el valor de la cadena del lado derecho:

```
$x = "Hola";
$x .= " mundo";       # es igual a escribir $x . " mundo"
print "$x \n";        # imprime: Hola mundo
```

En la siguiente tabla veremos los operadores de asignación contenidos en Perl, los que hemos visto, mas otros menos usados, de los que el lector puede descubrir su uso intuitivamente.

Operador	Uso	Equivalencia
=	$a = $b+1;	
+=	$a += 3;	$a = $a + 3;
-=	$a -= 3;	$a = $a - 3;
***=**	$a *= 2;	$a = $a * 3;
****=**	$a **= 2;	$a = $a ** 3;
/=	$a /= 3 + $b;	$a = $a / (3+$b);
%=	$a %= 8;	$a = $a % 8;
.=	$a .= ”hola”;	$a = $a.”hola”; ó $a = “${a}hola”;
>>=	$a >>= 1;	$a = $a >> 1;
<<=	$a <<= $b;	$a = $a << $b;
&=	$a &= (c += 3);	$c=$c+3; $a=$a & $c;
^=	$a ^= 2;	$a = $a^2;
|=	$a |= $c;	$a = $a\|$c;

4.1.6 Funciones definidas para manejo de cadenas de caracteres

Perl posee una serie de funciones definidas para el manejo de las cadenas de caracteres, las cuales nos ayudan muchísimo en las tareas de administración de datos.

A continuación nombraremos las funciones básicas para efectuar dicho tratamiento:

length

```
length($cadena).
```

Esta función nos permite conocer la longitud de una cadena de caracteres.
Por ejemplo:

```
$cadena = "hola";
$largo = length($cadena);     # $largo = 4
```

chop

```
chop($cadena).
```

Elimina el último caracter de la cadena y retorna dicho caracter.

Esta función se usa mucho para eliminar el caracter de salto de línea que contienen las cadenas que se introducen por teclado, o bien, de las líneas que se leen de un archivo, las cuales contienen también un salto de línea al final.

Ejemplo:

```
print "Teclea algo y pulsa enter:\n";

# En Perl, para capturar datos desde el teclado
# usamos la entrada estandard ó <STDIN>
$input = <STDIN>;     # $input contiene el "enter" , ó \n (salto de línea)

chop($input);    # ahora $input ya no contiene el último caracter: "\n"
```

Cabe mencionar que si tuviéramos la orden:

```
$ultimo = chop($input);       # $ultimo contendrá el salto de línea
                              # porque chop regresa el caracter extraído
```

chomp

Al igual que chop, elimina el último caracter de la cadena y retorna dicho caracter.
A diferencia de **chop**() que elimina cualquier caracter sea cual sea, **chomp**() sólo elimina el último caracter si éste se trata de un caracter de **salto de línea**. Por lo que siempre que necesitemos eliminar únicamente un caracter de nueva línea es más seguro utilizar **chomp**().

Los ejemplos anteriores funcionan adecuadamente, solo agregaremos ejemplos de la propiedad recursiva de **chomp**():

```
@array = ('Hola\n', 'mundo\n', 'Somos', 'todos\n');
chomp(@array);
```

En este ejemplo, hacemos un recorte recursivo de cada uno de los elementos del arreglo, de modo que cada uno de las variables queda sin el último caracter de salto de línea.

En el caso del tercer elemento, *Somos*, no le quita el último caracter (que sería la letra *s*), porque ese último caracter no es un salto de línea.

index

```
index(cadena, sub-cadena, [posición]).
```

Esta función retorna la posición de la primera ocurrencia de la sub-cadena dentro de la cadena indicada.

El parámetro posición indica el número de caracteres a partir del cual debe comenzar la búsqueda.

Si se omite este parámetro se considera que no se desea ignorar ningún carácter, es decir, se considera posición igual a 0.

Si no se encuentra la sub-cadena la función devuelve la posición desde la que comienza a buscar -1, esto es, el parámetro posición menos 1.

Ejemplo:

```
$cadena = "El automóvil";
$subcadena = "auto";                        # la sub-cadena a buscar
$pos = index($cadena, $subcadena);          # $pos = 4
$pos = index($cadena, $subcadena, 6);       # $pos = 5  (6-1)
```

rindex

```
rindex(cadena, sub-cadena, [posición]).
```

Esta función trabaja igual que *index()* salvo que retorna la última ocurrencia de la sub-cadena dentro de la cadena.

El parámetro posición indica el número de caracteres a partir del cual debe comenzar la búsqueda.
Si no se encuentra la sub-cadena la función devuelve [posición] menos 1.

Ejemplo:

```
$cadena = "Los celos del celoso";    # celos aparece dos veces
$sub = "celos";
$pos = rindex($cadena, $sub);        # $pos = 14;
```

substr

```
substr(cadena, inicio, [longitud]).
```

Esta función extrae una *sub-cadena* de la *cadena* dada, desde la posición indicada por *inicio*, hasta el número de caracteres indicado por *longitud*.

Si el parámetro *longitud* se omite se tomará la sub-cadena que va desde el parámetro *inicio* hasta el final de la cadena.
El primer carácter de una cadena es el que esta en la posición cero.

Ejemplo:

```
$cadena = "Universidad Autónoma de Nuevo León";
$sub = substr($cadena, 25, 5);       # $sub = "Nuevo"

# ahora no ponemos el parámetro longitud:
$sub = substr($cadena, 25);          # $sub = "Nuevo León"
```

uc

De upper case, devuelve la variable que se le pase como argumento, pero con todos los caracteres en mayúsculas.

Ejemplo:

```
$a = "Anastacio";
$b = uc($a);
print $b;      # imprime: ANASTACIO
```

lc

De lower case, es la función contraria a **uc**, devuelve una variable que se le pasa como argumento, pero con los caracteres en minúsculas.

Ejemplo:

```
$a = "ANASTACIO";
$b = lc($a);
print $b;      # imprime: anastacio
```

Funciones numéricas

abs(**$x**)	Devuelve el valor absoluto
cos(**$x**)	Coseno en radianes
exp(**$x**)	Exponencial
hex(**$x**)	Transforma un numero Hexadecimal a decimal
int(**$x**)	Devuelve la parte entera del número
log(**$x**)	Logaritmo natural (base e)
srand(**$x**)	Inicializa el generador de números aleatorios
rand(**$x**)	Devuelve un número real en el intervalo [0,$x)
sin(**$x**)	Seno en radianes
sqrt(**$x**)	Raíz cuadrada

Un solo ejemplo simple:

```
$a = 19.265257193;
$b = int $a;          # devuelve la parte entera de $a
print $b;             # imprime 19
```

O bien, mas elegante y menos líneas:

```
$a = 19.265257193;
print int $a;         # imprime 19
```

4.2 Funciones para Arrays

4.2.1 Función push

Agrega un elemento al final del array:

```
@array = ('carro', 'moto', 'lancha');       # 3 elementos iniciales

push (@array, 'camión');             # ahora @array tiene 4 elementos
                                     # y el último [3] es 'camión'

$otro = "avión";
push (@array, $otro);                # ahora @array tiene 5 elementos
                                     # y el último [4] es 'avión'

print $array[4], "\n";               # imprime: avión
```

4.2.2 Función pop

Al contrario de push(), **pop**() extrae y además devuelve el último elemento de un array:

```
@array = ('carro', 'moto', 'lancha', 'camión', 'avión');   # 5 elementos

pop(@array);                  # ahora @array queda con 4 elementos

$ultimo = pop(@array);        # ahora @array queda con 3 elementos
                              # y asigna el elemento extraído a la
                              # variable $ultimo

print "$ultimo \n";           # imprime: camión
```

4.2.3 Función shift

Extrae y devuelve el primer elemento de un array:

```
@array = ('carro', 'moto', 'lancha', 'camión', 'avión');   # 5 elementos

shift(@array);               # ahora @array queda con 4 elementos
                             # le hemos quitado 'carro'

$primero = shift(@array);    # ahora @array queda con 3 elementos
                             # y asigna el elemento extraído a la
                             # variable $primero

print "$primero \n";         # imprime: moto (era el primer elemento)
```

4.2.4 Función unshift

Al contrario de shift(), **unshift**() inserta un elemento al inicio de un array:

```
@array = ('lancha', 'camión', 'avión');     # 3 elementos

unshift(@array, "moto");            # ahora @array tiene 4 elementos

$otro = "carro";

unshift(@array, $otro);             # ahora @array tiene 5 elementos

print "$array[0] \n";               # imprime carro, el primer elemento
                                    # el cual fue insertado al último
```

4.2.5 Función split

Genera una lista ordenada por índices (o array), a partir de una cadena.

La cadena debe contener un patrón como un caracter o una cadena de caracteres, la cual servirá de ancla para separar cada elemento que formará el array.

Pero mejor veamos un ejemplo:

```
$cadena = "Casa==Edificio==Choza==Hotel";
@arraynuevo = split( "==", $cadena);

print $arraynuevo[1], "\n";          # imprime: Edificio
                                     # el segundo elemento del nuevo array
```

Note que el patrón que tomamos como base para separar los elementos del nuevo array, es el conjunto de caracteres doble signo igual: **==**.

Estos caracteres se eliminan al formar el array.

La función **split**(), soporta tres argumentos, el primero ya sabemos, es la cadena o cadenas que son nuestro patrón separador, el segundo es la cadena a partir de la cual generaremos el array, y existe un tercero que debe ser un número entero, el cual es opcional y con este le indicamos a **split**(), el límite de elementos que deseamos que contenga el array.

Esto hará que después de completar los elementos cargados al array, el resto de los separadores sea ignorado, formando todo el resto, el último elemento del array.

Ejemplo:

```
#/usr/bin/perl

$cadena = "Casa==Edificio==Choza==Hotel";
@arraynuevo = split("==", $cadena, 3);

print $arraynuevo[2], "\n";          # imprime: Choza==hotel
                                     # el último elemento del nuevo array
```

Note que Choza==Hotel no se separaron por el patrón elegido (==), pues solo escogimos 3 elementos máximos, ignorando el resto de las cadenas separadoras.

4.2.6 Función join

Toma una lista ordenada por índices o array, y concatena los elementos de esa lista o array, mediante una o unas cadenas.

Es realmente, lo contrario de **split()**:

Ejemplo:

```
#/usr/bin/perl
@array = ("Casa", "Edificio", "Choza", "Hotel");

$cadenanueva = join("--", @array);

print $cadenanueva, "\n";     # imprime: Casa--Edificio--Choza--Hotel
```

Concatenamos todos los elementos del array *@array* interponiendo un separador que en

este caso, usamos la cadena doble guión **--**.

El separador puede ser cualquier caracter, incluso el espacio, o una cadena vacía: **""**.

4.2.7 Función sort

Ordena los elementos de un array en forma ascendente.

Ejemplo:

```
#!/usr/bin/perl

@array = (5, 2, 1, 3, 4);

@array = sort(@array);              # ahora @array = (1, 2, 3, 4, 5)

print $array[4], "\n";              # imprime 5
```

Lo mismo pasa con caracteres alfabéticos:

```
#!/usr/bin/perl

@array = ('a', 'e', 'b', 'd', 'c', 'B');

@array = sort(@array);              # ahora @array = (B, a, b, c, d, e)

print $array[5], "\n";              # imprime e

print $array[0], "\n";              # imprime B
```

Note que las Letras Mayúsculas tienen precedencia sobre las minúsculas.

Rogamos al lector buscar la documentación de sort en:
http://perldoc.perl.org/functions/sort.html
y que trate de comprender bien esta función, pues es muy útil en el análisis de datos.

4.2.8 Función reverse

Invierte el orden completo de los elementos de un array.

Ejemplo:

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

```
@array = (5, 2, 1, 3, 4);

@array = reverse(@array);            # ahora @array = (4, 3, 1, 2, 5)

print $array[0], "\n";               # imprime 4
```

Y si se nos ocurre mezclar **reverse**() con el anterior operador estudiado, **sort**():

```
#/usr/bin/perl

@array = (5, 2, 1, 3, 4);

@array = reverse( sort(@array) );    # ordenamos y después invertimos
                                     # ahora @array = (5, 4, 3, 2, 1)

print $array[0], "\n";               # imprime 5
```

- - -

Adicionalmente, podemos reducir expresiones, como lo hacemos en matemáticas, solo que en Perl, se trata de aplicar nuestra lógica:

El último script lo podemos reducir a:

```
#/usr/bin/perl
print reverse (sort(5, 2, 1, 3, 4)), "\n";
```

Aplicamos la función **print** a la función **reverse**, y esta se aplica a dos elementos, la función **sort** y el salto de línea, la función **sort** aplica sus bondades sobre una lista de elementos.

El resultado... se imprime 54321, y un salto de línea...

4.2.9 Función $#

Este operador nos permite conocer el último índice de una matriz del tipo array referenciado anónimo bidimensional.

Examinemos el siguiente script que ya lo escribiremos completo:

```
#!/usr/bin/perl

$array = [                    # creamos el array referenciado
  ['uno', 'dos', 'tres'],           # primera línea o sub-array
  ['cuatro', 'cinco', 'seis'],      # segunda línea o sub-array
  ['siete', 'ocho', 'nueve']        # tercera línea o sub-array
];

print $#{$array}, "\n";       # imprime 2, que es el último índice de líneas
                              # o sea, que contiene del 0 al 2

print $#{$array} + 1, "\n";   # imprime 3, que es la cantidad real de líneas

exit;                         # salimos elegantemente del script
```

Si nuestra variable referenciada contiene tres elementos sub-arrays, esto nos va a devolver 2, que es el último índice del sub-array contenido en la matriz.

4.2.10 Función map

Otra función importante que podemos usar es **map**, la cual recorre iterativamente los elementos de un array o lista, y puede cambiarlos si lo deseamos.

La sintaxis es:

```
map { INSTRUCCIONES_POR_CADA_ELEMENTO } @array;
```

Esta función también devuelve el resultado a una variable similar:

```
@otroarray = map { INSTRUCCIONES_POR_CADA_ELEMENTO } @array;
```

Ejemplo:

```
@array = (1, 2, 3, 4, 5);
map { $_ += 10 } @array;      # ahora @array = (11, 12, 13, 14, 15)
```

4.2.11 Función grep

Devuelve una lista que contiene los elementos de un array donde la expresión es verdadera.

Sintaxis:

grep (*EXPRESIÓN*, @array);

Ejemplo:

```
@a = ("a1", "a2", "b1", "b2", "c1", "c2");
@b = grep ( /^b/, @a );       # El lector no ha leído sobre esta
                              # expresión, mas adelante lo veremos

map { print $_, " " } @b;     # imprime: b1 b2 usando la función anterior
```

Ya que aún no hemos estudiado las expresiones regulares, en este caso le pediremos al lector que imagine que le pedimos a grep que busque los elementos del array que comienzan con la letra **b**.
Posteriormente usted podrá regresar a entender bien esta función.

4.3 Funciones para Hashes

4.3.1 Función keys

Devuelve una lista de las llaves de una lista asociativa o Hash.

Ejemplo:

```
#!/usr/bin/perl

%hash = (
  'carro' => 'azul',
  'moto' => 'roja',
  'camion' => 'verde',
  'avion' => 'amarillo'
);

@arraykays = keys(%hash);     # @arraykeys = (carro, moto, camion, avion)

print $arraykeys[1], "\n";    # imprime moto
```

Recordemos que un hash está formado por una lista de pares *llave – valor.*

4.3.2 Función values

Devuelve una lista de los valores de una lista asociativa o Hash.

Ejemplo:

```
#!/usr/bin/perl

%hash = (
  'carro' => 'azul',
  'moto' => 'roja',
  'camion' => 'verde',
  'avion' => 'amarillo'
);

@lsvalues = values(%hash);    # @lsvalues = (azul, roja, verde, amarillo)

print $lsvalues[2], "\n";     # imprime verde
```

4.3.3 Función delete

Suprime o elimina elementos de una lista asociativa o Hash.

Ejemplo:

```
#!/usr/bin/perl

%hash = (
  'carro' => 'azul',
  'moto' => 'roja',
  'camion' => 'verde'
);

delete $hash{'moto'};         # eliminamos un elemento del hash

@arraykays = keys(%hash);     # @arraykeys = (carro, camion)

print $arraykeys[1], "\n";    # imprime camion
```

4.3.4 Función each

Este operador recorre iterativamente todos los pares llave-valor de una lista asociativa o Hash, permitiendo acceder recursivamente a cada par mediante el uso de estructuras de control.

La sintaxis operativa es:

```
( llave, valor ) = each ( hash )
( $llave, $valor ) = each ( %hash ) ;
```

No podemos hasta este momento dar ejemplos de este operador ya que implica usar métodos que aún no hemos estudiado.

Por este motivo, dejaremos la explicación con ejemplos para una parte mas adelante, en la sección de “La instrucción while” del siguiente capítulo.

CAPÍTULO 5
Estructuras de control

5. Estructuras de control en Perl

Al inicio del libro hablamos de que un script de Perl es una susceción de instrucciones.

Estas instrucciones se ejecutan en orden una tras otra hasta que termine el script.

Sin embargo, hay métodos de controlar el flujo de un script para que se ejecuten de diferentes maneras las instrucciones que tengamos a la vista.
Unas si, otras no según un criterio, otras mas de forma repetitiva hasta que se cumpla alguna condición establecida por nosotros y prosigamos con el curso "normal" del script.

Para controlar el flujo de un script tenemos en Perl varias estructuras para este efecto.

Bloque de instrucciones

La primera cosa que debemos tomar en cuenta es que podemos ejecutar un **Bloque de Instrucciones**, el cual está delimitado por los signos de llaves **{ }**.
Y todas las instrucciones dentro de este bloque se ejecutan una a una en secuencia normal como se ejecutan en todo el script.

```
{
  primera instrucción...
  segunda instrucción...
  ...
  última instrucción...
}
```

Este **bloque de instrucciones** es aceptado como si fuera una simple instrucción, esto es, que se ejecuta en el orden de una simple instrucción según el orden general del script.

5.1 La función if

Es una función que ejecuta un bloque de instrucciones, mientras una condición establecida sea evaluada y resulte verdadera.

Una condición verdadera tiene que tener un valor binario diferente de cero (0).

La sintaxis es la siguiente:

```
if  ( condición = verdadero )
{
        ... Bloque de instrucciones a ejecutar (si condición = verdadero).
}
```

Recodemos que por defecto, en Perl una variable vacía tiene un valor cero o falso, y en caso contrario, si una variable no es igual a vacío o cero, es verdadero.

Vacio => falso
0 => falso
no vacío => verdadero
diferente de 0 => verdaddero

La misma sintaxis la tenemos virtualmente de esta manera:

```
if  ( 1 )
{
        ... Bloque de instrucciones a ejecutar.
}
```

Donde lo que colocamos dentro del paréntesis resulta ser un 1 o diferente de cero, y al final de cuentas, verdadero.

En perl, podremos tener diferentes maneras de evaluar una condición.

Recordemos los operadores relacionales de la sección 4.1.2, los cuales evalúan una expresión para retornar un valor de falso o verdadero.

Pues estos operadores cumplen su función perfecta dentro de las **funciones** como **if**.

Hagamos un pequeño ejemplo:

```
#!/usr/bin/perl

$var = "";     # variable vacía = 0 = falso

if ( $var ne "" ) {          # si $var "no es igual" a vacío
        print $var, "\n";
}
```

En este caso no se imprime absolutamente nada, ya que la condición es falsa, porque $var si es igual a vacío.

Ahora asignemos un valor a $var.

```
#!/usr/bin/perl

$var = "Hola mundo...";       # variable no vacía

if ( $var ne "" ) {           # si $var "no es igual" a vacío
        print $var, "\n";
}
```

Ahhhhh, ahora si imprimimos "**Hola mundo...**"

Recordemos unas cuantas líneas atrás, donde decíamos que se podrán ejecutar las instrucciones del bloque si tenemos una valor verdadero, y que una variable **no vacía** es igual a verdadero.

Podemos la misma instrucción escribirla así:

```
#!/usr/bin/perl

$var = "Hola mundo...";       # variable no vacía = 1 lógico = verdadero

if ( $var ) {          # si $var existe = diferente de 0 = verdadero
        print $var, "\n";
}
```

Imprimiendo "**Hola mundo...**" porque la evaluación de la cadena resultó ser diferente de cero (o no vacío) o verdadero.

Esto nos pone a pensar (si no nos sale humo de nuestra cabeza), que tal si declaramos nuestra variable con un valor de cadena pero que esta sea cero:

```
#!/usr/bin/perl

$var = "0";    # variable escalar con contenido de caracter 0
               # ojo! por las comillas no es un número, es una cadena

if ( $var ) {          # si verdadero o 1 lógico
        print $var, "\n";
}
```

Resultó como pensamos, un escalar de caracter "0" es igual a cero numérico, o sea, falso.

Nuestra variable se transforma dependiendo del contexto donde se esté usando, y si al principio declaramos la variable en un contexto escalar de caracteres, el uso que le damos es alrededor de un contexto numérico, por lo tanto, la variable vale cero numérico o falso.

En este caso no imprime nada, ni el "0" porque la condición evaluada devuelve 0 o falso,

aún y cuando la variable no esté vacía inicialmente por el escalar asignado "0".

Sin embargo, si en lugar de asignarle un valor de un solo cero, le damos un valor de doble cero como cadena, este no valdrá 0, sino que ya es una cadena de texto no vacía.

Veamos:

```
#!/usr/bin/perl

$var = "00";   # variable escalar con contenido de caracteres 00
               # ojo! Ya es una cadena de texto no vacía

if ( $var ) {         # si $var es cadena no vacía o 1 lógico
       print $var, "\n";
}
```

Esto si imprimirá algo, el valor de **$var** que es **00** ya que la variable **$var** es un escalar no vacío y por lo tanto, igual a un 1 lógico y la evaluación devuelve verdadero, por lo que se ejecuta lo que está dentro del bloque de instrucciones.

5.1.1 Else

Ahora, que pasaría si nos interesa ejecutar otro bloque de instrucciones cuando la condición resulte falsa...
Esto es, si es verdadero, ejecuta esto, si es falso ejecuta esto otro...

Podremos usar la función **else** para este resultado.

Sintaxis:

```
if ( condición = verdadero )

{
       ... bloque ejecutable si la condición devuelve verdadero
}

else

{
       ... bloque ejecutable si la condición no devuelve verdadero
}
```

Veamos un ejemplo:

```
#!/usr/bin/perl

$var = "";

if ( $var ) {                  # si $var "no es igual" vacío o 0
        print $var, "\n";
}
else
{
        print "Nada que imprimir \n";
}
```

Esto imprime desde luego "**Nada que imprimir**" (y un salto de línea), porque la variable **$var** estaba vacía y se ejecuta lo que se encuentre dentro del bloque siguiente a **else**.

En la práctica podemos colocar el cierre del primer bloque, la función else y la apertura del segundo, en la misla línea para ahorrarnos espacio en líneas y las cosas sean mas legibles:

```
#!/usr/bin/perl

$var = "Hola Mundo!!";

if ( $var ) {
        print $var, "\n";
} else {
        print "Nada que imprimir \n";
}
```

Imprime con el ejemplo anterior, el "Hola mundo!!" de la variable **$var**.

5.1.2 Elsif

Cuando tenemos muchas funciones **if**() subsecuentes y están relacionadas, es decir, se tienen dos o mas diferentes opciones, podemos escribirlo así:

```
if ( Condicion1 ) {
        Bloque de instrucciones 1
        cuando Condicion1 devuelve verdadero
} elsif ( Condicion2 ) {
        Bloque de instrucciones 2
        cuando Condicion2 devuelve verdadero
        y Condicion1 devuelve falso
} elsif ( Condicion3 ) {
        Bloque de instrucciones 3
        cuando Condicion3 devuelve verdadero
        y Condicion2 devuelve falso
```

```
} elsif ( Condicion4 ) {
        Bloque de instrucciones 4
        cuando Condicion4 devuelve verdadero
        y Condicion3 devuelve falso
} else {
        Bloque de instrucciones al devolver falso todo
}
```

5.1.3 Sintaxis casi real

Perl está diseñado por un lingüista, Larry Wall, y es por esto que nuestro lenguaje favorito es capaz de soportar estructuras sintácticas muy parecidas a la forma como hablamos (bueno... en inglés).

La sintaxis de if:

```
if  ( condición = verdadero )
{
        ... Bloque de instrucciones a ejecutar (si condición = verdadero).
}
```

Es posible escribirla de otra forma:

```
$var = "algo" if  condición = verdadero
```

Ejemplo:

```
#!/usr/bin/perl
$var = "Hola Mundo!!";        # asignamos una cadena a $var

print "$var \n" if $var;      # se imprime $var si posee un valor
```

Esto imprime desde luego, *Hola Mundo* (si **$var** tiene valor = verdadero).

Si **$var** no tiene valor, es decir, vale un cero lógico o falso, no se ejecuta la instrucción con la función **print**.

5.2 La función unless

Ahora, que pasa si no queremos ejecutar nada cuando la condición devuelva verdadero, solo cuando la condición devualva falso.

En teoría **sería** así:

```
#!/usr/bin/perl

$var = "Hola Mundo!!";

if ( $var ) {                 # si $var "no es igual" vacío o 0

        # NO quiero hacer nada

} else {
        print "Nada que imprimir \n";
}
```

Para solucionar esto, existe la función **unless**, que ejecuta un bloque de instrucciones cuando la evaluación de una condición devuelva falso.

El mismo ejemplo usando la función correcta:

```
#!/usr/bin/perl

$var = "";

unless ( $var ) {             # si $var "es igual" vacío = 0 = falso

        print "Nada que imprimir \n";
}
```

Note que puede ser lo contrario a la función **if**().
Cuando la variable posee contenido no queremos hacer nada, así que si le asignamos un valor a $var, no imprimimos nada. Esto es, no se ejecutará el bloque siguiente a **unless**().

Por último agregamos que **unless**() también posee compatibilidad con **else**.
Dejamos a su propio criterio experimentar con esto y usarlo en lugar de la función **if**().

5.3 La función while

Existen en Perl también unas funciones que ejecutan un bloque de instrucciones en forma repetitiva, mientras una condición sea verdadera.

El bucle o ciclo se repite indefinidamente hasta que la condición devuelva falso.

La función **while**() es simple y requiere de una condición en igual forma que **if**().

Sintaxis:

```
while ( condición = verdadero ) {
       ... Se ejecuta este bloque una y otra vez
}
```

Ejemplo:

```
#!/usr/bin/perl

$cont = 0;

while ( $cont <= 10 ) {
       print "El contador es: ", $cont, "\n";

       $cont ++;             # recordemos el operador aritmético
                             # de incremento del punto 4.1.1
}

print "Ya salimos del ciclo \n";

exit;
```

En este ejemplo vemos que comenzamos el ciclo con un valor de la variable $cont igual a cero, y con cada vuelta del ciclo, incrementamos el valor de $cont en uno, así $cont vale 1, después 2, luego 3, y así hasta que vale 10, en ese caso, la condición ya es falsa, y **while**() termina el ciclo, continuando con el curso normal del script.

Mientras esto suscede, ejecutamos el bloque de instrucciones dentro de las llaves a cada ciclo de while, que en este caso solo refieren a imprimir la misma variable $cont, para efectos de análisis.

El resultado si ejecutamos el script sería algo como esto:

```
user@host # ./script.pl
El contador es: 0
El contador es: 1
El contador es: 2
El contador es: 3
El contador es: 4
El contador es: 5
El contador es: 6
El contador es: 7
El contador es: 8
El contador es: 9
El contador es: 10
Ya salimos del ciclo
user@host #
```

Existe una característica que hace de while un bucle muy especial.
Y es que while puede recorrer toda una lista de elementos, aunque no sea una lista de array.

De la explicación que dejamos pendiente del *operador each* del punto 4.3.4, podremos recorrer uno a uno cada par *llave-valor* de un Hash.

Veamos:

```
#!/usr/bin/perl

%hash = (              # Declaramos nuestro Hash
  'carro' => 'azul',
  'moto' => 'roja',
  'camion' => 'negro',
);

while ( ($llave, $valor) = each(%hash) ) {  # recorremos el Hash
  print $llave . " == " . $valor . "\n";    # imprime cada par llave-valor
}
```

El resultado impreso en la pantalla será algo parecido a esto:

```
user@host # ./script.pl
carro == azul
moto == roja
camion == negro
user@host #
```

En otro aspecto, si acaso declaramos artificialmente verdadera la condición que evalúa while, el ciclo se ejecutará indefinidamente hasta que no cancelemos el proceso del script desde afuera.

Ejemplo:

```
#!/usr/bin/perl

$cont = 0;

while ( 1 ) {
        print "El contador es: ", $cont, "\n";

        $cont ++;             # recordemos el operador aritmético
                              # de incremento del punto 4.1.1
}

print "Ya salimos del ciclo \n";
```

Este bucle nunca va a imprimir el ya visto: *Ya salimos del ciclo*, puesto que nunca sale del bucle, además de que los incrementos se harán hasta el infinito.

Si estamos en GNU/Linux, con oprimir las teclas *control + C* cancelamos el proceso actual, saliendo del script completo.

5.4 La función for

Otro bucle que se ejecuta iterativamente mientras una condición sea verdadera, es **for**()

Sintaxis:

```
for ( inicio; condición; incremento ) {
    intrucción_1;
    intrucción_2;
    intrucción_3;
}
```

Regresando al bucle **while**() del punto anterior:

```
#!/usr/bin/perl
$i = 0;
while ( $i <= 10 ) {
       print "El contador es: ", $i, "\n";
       $i ++;                # recordemos el operador aritmético
                             # de incremento del punto 4.1.1
}
exit;
```

Esto mismo lo podemos escribir con **for**() de la siguiente manera:

```
#!/usr/bin/perl

for ($i = 0; $i <= 10; $i++) {
    print "El contador es: ", $i, "\n";
}

exit;
```

Usando menos líneas, imprimimos exactamente lo mismo que con el uso de while.
No debe hacer falta explicaciones para esta función, y si es así, creo que debemos repasar de nuevo lo que llevamos del libro.

El operador **for**() también es capaz de procesar una lista como **while**():

```
#!/usr/bin/perl

for $i ( 0 .. 9 ) {           # rango del 0 al 9
        print $i, "\n";       # imprime del 0 al 9 con salto de línea
}

exit;
```

Pero para este uso mas completo existe la función **foreach**(), que veremos a continuación.

5.5 La función foreach

La construcción foreach() recorre iterativamente cada elemento de una lista dada.
La sintaxis lo explica bien:

```
foreach $elemento ( de, la, lista, de, elementos ) {
    bloque de instrucciones...
}
```

Un ejemplo simple:

```
#!/usr/bin/perl

@a = (1,2,3,4,5);
foreach $i ( @a ) {
    print $i, "\n";
}
```

La función **foreach**() es capaz de alterar cada elemento de la lista cuando la recorre.

```
#!/usr/bin/perl

@a = (1,2,3,4,5);

foreach $i ( @a ) {
    $i *= 3, "\n";    # ahora @a = (3, 6, 9, 12, 15)
}
```

5.6 La función until

Esta función es poco vista en Perl, sin embargo no es menos importante que las demás, ya que todo en Perl tiene aplicaciones específicas.

Ejecuta un bloque de instrucciones mientras la condición resulte falsa. Es, aparentemente,

lo contrario de **while**(), saliendo del bucle cuando la condición regrese verdadero.

Sintaxis:

```
until ( Condición = falso ) {
    Bloque de instrucciones...
}
```

Con un ejemplo lo ilustramos:

```
#!/usr/bin/perl

$i = 0;

until ( $i >= 10 ) {          # mientras $i no sea igual o mayor a 10
       print "El contador es: ", $i, "\n";          # se imprime $i
       $i ++;                 # incremento de 1
}

exit;
```

Esto imprimirá del 0 al 9 porque $i = 10 ya devuelve falso a la función.

5.7 Bifurcaciones para los bucles

En algunos momentos desearemos tener un control un poco mas estricto de lo que pase dentro de una función.

Supongamos que tenemos un array de mas de 100,000 elementos obtenidos de cualquier parte de un servidor, y debemos encontrar si alguno de los elementos es el que buscamos.

Que tal si encontramos el elemento con el índice 225... Tendremos que esperar a que el bucle recorra el resto de los elementos para continuar con el script ?

La respuesta es no, hay métodos para salir de un bucle o ciclo, así como para continuar con él.

5.7.1 La salida last

Sale definitivamente de un bucle. Marca el ciclo actual como el último.

Ejemplo:

```
#!/usr/bin/perl

$i = 0;
while ( $i <= 10000 ) {       # se inicia el ciclo hasta 10,000
        print $i, "\n";
        $i ++;                # recordemos el operador aritmético
                              # de incremento del punto 4.1.1
        if ( $i == 10 ) {
                last;         # salimos del bucle cuando $i = 10
                              # No esperamos a que pasen los 10,000
        }
}
exit;
```

Esto imprime del 0 al 9, porque cuando $i sea igual a 10, sale del bucle sin esperar a que se realice otro ciclo debido a la bifurcación **last**.

5.7.2 La función next

Lo contrario de **last**, **next** continua con el ciclo.
Esta función está implícita en cualquier bucle, solo que en ocasiones puede ser necesario declararlo explícitamente por necesidades específicas del script.

Ejemplo:

```
#!/usr/bin/perl

$i = 0;
while ( $i <= 10000 ) {
        print $i, "\n";
        $i ++;                # recordemos el operador aritmético
                              # de incremento del punto 4.1.1
        if ( $i == 10 ) {
                last;         # salimos del bucle cuando $i = 10
                              # No esperamos a que pasen los 10000
        } else {
                next;         # aprendamos como funciona
                              # talvez lo necesitemos
        }
}
exit;
```

Hace exactamente lo mismo que el ejemplo anterior.

5.7.3 Etiquetas

Hay ocaciones en que debamos tener un bucle dentro de otro bucle, por ejemplo, un ciclo **while** dentro de un ciclo **foreach** o viceversa.
Sin embargo dentro del bucle interior colocamos una función **if** para poder salir no solo del bucle interior, sino también del bucle externo.

Como **last** solo sale del bucle donde está, es necesario colocar etiquetas para "marcar" la instrucción para hacer una referencia de espacio.

Las etiquetas no tienen ningún prefijo especial de puntuación, solo se escriben.
Así podremos usar la función **last** destinando la salida del bucle marcado por la etiqueta:

```
last ETIQUETA;
```

Ejemplo simple:

```
ETIQUETA: while (condición = 1) {
    instrucción;
    instrucción;
    instrucción;
    if (condición adicional) {
        last ETIQUETA;
    }
}
```

Ejemplo compuesto (desde *Learning* Perl ©):

```
#!/usr/bin/perl

EXTERNO: for ($i = 1; $i <= 10; $i++) {
    INTERNO: for ($j = 1; $j <= 10; $j++) {
        if ($i * $j == 63) {
            print "$i veces $j es igual a 63!\n";
            last EXTERNO;
            }
        if ($j >= $i) {
            next EXTERNO;
        }
    }
}
exit;
```

El primer bucle recorre iterativamente los números del 1 al 10 y el segundo lo hace igualmente, por lo que las multiplicaciones de $i * $j comienzan desde (1,1), (2,1), (2,2), (3,1), (3,2), (3,3), (4,1), y así sucesivamente hasta encontrar la combinación exacta cuyo

resultado sea 63 (7 * 9) y entonces, sale no solo del bucle interno, sino también del bucle externo para no ejecutar el resto de los ciclos, por medio de la instrucción:

```
last EXTERNO;
```

Si ejecutamos el script debe imprimir algo parecido a esto:

```
user@host:# ./test.pl
9 veces 7 es igual a 63!
user@host:#
```

CAPÍTULO 6
Expresiones regulares

6. Expresiones regulares

Una expresión regular es un mecanismo que busca un patrón dentro de una cadena de texto.

Las expresiones regulares son usadas por muchos programas como los comandos de Unix grep, sed, awk, ed, vi, emacs, etc, y cada programa desde luego tiene sus variantes en su sintaxis.

En Perl, podemos delimitar una expresión regular generalmente encerrando la o las cadenas entre slashes o barras inclinadas:

```
/cadena/
```

Y esto significa que dentro de la cadena dada, queremos encontrar el patrón indicado dentro de los slashes.

6.1 Expresiones regulares de comparación

La forma de uso mas básica de las expresiones regulares es precisamente la comparación, y se trata de encontrar una cadena dentro de otra.

Para esto usamos siempre la sintaxis:

```
$cadena =~ /patrón/;
```

Usando el signo de igual seguido de un tilde.

Veamos un ejemplo en un script:

```
#!/usr/bin/perl

$datos = "Cristobal Colon descubrió América";

if ( $datos =~ /Colon/ ) {    # Buscamos "Colon" dentro de $datos
                              # y devuelve verdadero si lo contiene

    print "Si contiene \"Colon\" \n";       # no olvidemos el \ para las "

} else {
    print "No contiene \"Colon\" \n";
}
```

Esto desde luego, imprime *Si contiene "Colon"*.

Si buscamos que una cadena contenga por ejemplo, un semicolon (**;**), es necesario usar, como siempre, el backslash:

```
$datos =~ /hola\;/;
```

Para no confundirlo con el semicolon del final de instrucción.

Hay símbolos que representan diferentes cosas a las del caracter específico, y son los caracteres reservados:

```
* + $ ^ | [ ] ( ) { } - . \w \s \d \n \
```

El asterisco se usa para buscar un caracter repetido dentro de una cadena, y como mínimo debe encontrar 0 ocurrencias del mismo, es decir, puede que haya, puede que no... 8{D

```
$cdn =~ /ab*c/;
```

En esta expresión buscamos:
Primero un caracter a, seguido de b's mínimo cero hasta *n* veces, y seguido de una c.

La b no es indispensable que exista en la cadena:

```
#!/usr/bin/perl

$abc = "ac";          # podemos poner $abc = "abbbc" o $abc = "abc"

if ( $abc =~ /ab*c/ ) {
    print "Sí contiene ac, o abc, o abbbc \n";
}
exit;
```

Si la b fuera indispensable que exista en la cadena, usaremos el signo **+**:

```
#!/usr/bin/perl

$abc = "ac";          # podemos poner $abc = "abbbc" o $abc = "abc"

if ( $abc =~ /ab+c/ ) {              # si contiene una o mas b's
       print "Sí contiene abc, o abbbc \n";
} else {
       print "No contiene abc, o abbbc \n";
}
exit;
```

En este caso le pedimos a la expresión que nos devuelva verdadero si encontramos un caracter a, seguido de b's mínimo una hasta *n* veces, y seguido de una c.
El resultado al imprimir será: *No contiene abc, o abbbc*.

Ahora, cuando el patrón buscado debe estar el inicio de la cadena, usaremos el signo de **^**:

```
#!/usr/bin/perl

$abc = "abcxyz";

if ( $abc =~ /^abc/ ) {              # buscamos al inicio de la cadena
        print "Sí contiene abc al inicio \n";
} else {
        print "No contiene abc al inicio \n";
}
exit;
```

Esto imprimirá *Si contiene abc al inicio* porque el patrón buscado sí se encuentra al principio de la cadena $abc.

Y si deseáramos buscar el patrón al final de la cadena, en este caso usaremos el **$** al final del patrón buscado para indicar nuestros deseos:

```
#!/usr/bin/perl

$abc = "abcxyz";

if ( $abc =~ /xyz$/ ) {              # buscamos al final de la cadena
        print "Sí contiene xyz al final \n";
} else {
        print "No contiene xyz al final \n";
}
exit;
```

Y claro que imprimirá *Sí contiene xyz al final*, porque es el patrón que buscamos y éste se encuentra al final de la cadena $abc.

Y que tal si queremos una cosa o la otra ?
En este caso usaremos el operador booleano **or**, que en nuestras expresiones regulares tiene la sintaxis:

```
/uno|otro/
```

Así, nuestro ejemplo lo podemos modificar y explicarlo como: *si contiene abc al inicio **o** xyz*

al final de la cadena ... :

```
#!/usr/bin/perl

$abc = "abcdefghijklmnopqrstvwxyz";

if ( $abc =~ /^abc|xyz$/ ) { # buscamos al inicio o al final de la cadena

        print "Sí contiene abc al inicio, o xyz al final \n";

} else {
        print "No contiene abc al inicio, o xyz al final \n";
}
exit;
```

Al ejecutar el script nos imprime: *Sí contiene abc al inicio, **o** xyz al final*, que es lo que buscábamos. Se cumplen ambas cosas.

Cuando tengamos que especificar un rango de caracteres, podremos hacer uso de los corchetes, [abc] significaría: cualquier caracter entre *a*, *b* o *c,* es decir, pueden ser cualquiera de estos tres... solo uno.

Ejemplo:

```
#!/usr/bin/perl

$abc = "abcdef";

if ( $abc =~ /[aqz]/ ) {      # buscamos a, q o talvez z

        print "Sí contiene a, q ó z \n";

} else {
        print "No contiene a, q ó z \n";
}
exit;
```

En este script, como la cadena contiene cualquier caracter a escoger entre *a*, *q* ó *z*, se imprime *Sí contiene a, q ó z*.

Si a $abc le quitamos la *a* y la cambiamos por *g* o *z*, dará el mismo resultado, cuando la cadena no contenga ninguno de estos tres caracteres, devolverá falso.

También podemos especificar un rango completo mediante el uso del guión:

```
#!/usr/bin/perl

$abc = "abcdef";

if ( $abc =~ /[a-zA-Z]/ ) {   # buscamos rangos, de la "a" a la "z"
                              # minúscula o mayuscula
        print "Sí contiene desde a hasta la z \n";

} else {
        print "No contiene desde a hasta la z \n";
}
exit;
```

Aquí se imprime *Sí contiene a, q ó z*, porque estamos buscando cualquier caracter entre la a y la z y también entre la A y la Z dentro de la cadena **$abc**.

Otros operadores de expresiones regulares son:

\w = cualquier caracter alfanumérico y el guión bajo
\s = un espacio
\d = dígitos (números)
\n = salto de línea
\t = tabulación
\r = retorno de carro
\f = salto de página
. = cualquier cosa

Ejemplos varios:

Si queremos encontrar cuaquier vocal:

```
/[aeiouAEIOU]/
```

Esto lo podemos escribir de esta otra forma:

```
/[aeiou]/i
```

Note al final después del slash de cierre que usamos la letra *i*, esto para decirle a las expresiones regulares que no nos importa si es mayúscula o minúscula.

Cualquier dígito:

```
/[1234567890]/
```

Esto lo podemos escribir mas fácilmente así:

```
/[0-9]/       # es un rango del 0 al 9, cualquier dígito
```

Cualquier letra o dígito:

```
/[a-zA-Z0-9]/  # son rangos, cualquier letra o dígito.
```

Cualquier conjunto de... un espacio, luego letras o dígitos, y después otro espacio:

```
/\s[a-zA-Z0-9]\s/
```

El conjunto de... la cantidad que sea de espacios, seguido de letra o dígito, y después otro espacio:

```
/\s+[a-zA-Z0-9]\s/
```

Note el signo de + para declarar que son *uno o mas espacios*.

El conjunto de... espacios (la cantidad que sea), letras o dígitos (la cantidad que sea), y después otros espacios (la cantidad que sea):

```
/\s+[a-zA-Z0-9]+\s+/
```

Lo que sea, la cantidad que sea:

```
/.*/
```

Las letras abc, seguidas de cualquier cosa, y después xyz:

```
/abc.*xyz/
```

Equivalencias entre caracteres predefinidos y patrones creados:

```
\w = [a-zA-Z0-9_]
\s = [ ]
\d = [0-9]
```

Usando un delimitador diferente.

Cuando no queremos confundirnos con los slashes (aplicando el backslash), por ejemplo:

```
/^\/usr\/etc/
```

Es mejor cambiar el delimitador (aquí usamos **!** en lugar del slash tradicional):

```
m!^/usr/etc!
```

O cualquier otro (ahora usamos **#**):

```
m#^/usr/etc#
```

Note la ***m*** antes de comenzar el delimitador especial (de match).

6.1.1 Memoria de las expresiones

Dentro de las expresiones regulares, podremos encerrar entre paréntesis algunos grupos de cadenas las cuales deseamos usar o extraer para procesarlas posteriormente.
Estas pequeñas partes retenidas en la memoria de la expresión, las podemos llamar posteriormente como variables especiales, sucesivamente: *$1, $2, $3... $n*.

Ejemplo:

```
#!/usr/bin/perl

$var = "La vuelta al mundo en 80 días";

$var =~ /\w+\s([a-zA-Z]+)\s.*/;

$palabra1 = $1;       # $1 es lo que está entre paréntesis: ([a-zA-Z]+)
                      # cualquier cantidad de letras mayúsculas o minúsculas
print $palabra1, "\n";       # imprime: vuelta
```

Si extraemos mas cadenas memorizadas:

```
#!/usr/bin/perl

$var = "La vuelta al mundo en 80 días";

$var =~ /^[a-zA-Z]+\s([a-z]+)\s\w*\s([a-z]+)\s.*/;

$p1 = $1;             # es lo que está en el primer paréntesis
$p2 = $2;             # el segundo paréntesis

print $p1 . " " . $p2 . "\n";        # imprime: vuelta mundo

exit;
```

Ahora podemos usar todos estos datos para ejemplificar y resolver un pequeño problema.

- - -

Problema:
Mediante el comando Unix *host dominio*, podemos saber la IP de un nombre de dominio cualquiera.
Debemos extraer la IP del dominio google.com.

Si en línea de comandos ejecutamos:

```
user@host # host google.com
```

Nes devuelve algo así:

```
user@host:# host google.com
google.com has address 74.125.45.100
google.com has address 74.125.67.100
google.com has address 209.85.171.100
google.com mail is handled by 10 smtp1.google.com.
google.com mail is handled by 10 smtp2.google.com.
google.com mail is handled by 10 smtp3.google.com.
google.com mail is handled by 10 smtp4.google.com.
user@host:#
```

(Raro, no ???)

Entes de proseguir, explicaremos como interactúa Perl con comandos del sistema operativo.
Existen algunas maneras de ejecutar un comando del OS y tomar el resultado en una variable.

Primer método, la función **system**:

```
$resultado = system("ls -lrt");
```

En este comando simplemente listamos el contenido del directorio actual.

Segundo método, el apóstrofe (`):

```
$resultado = `ls -lrt`;

print $resultado, "\n";
```

Recordemos que este libro está enfocado a interactuar con Servidores de alto rendimiento

por lo que asumimos que éstos usan algún sabor de Unix o GNU/Linux.

Regresemos a nuestro problema original, la salida del comando *host* nos servirá para analizar las líneas de salida y generar nuestro código:

```
#!/usr/bin/perl

$cmd = `host google.com`;     # mecanismo de ejecución de un comando
                              # a nivel del Sistema Operativo: `command`.

@arrayCmd = join(/\n/, $cmd);        # separamos cada línea y las colocamos
                                     # como elementos de un array

# ahora analizamos el array en un bucle y buscamos patrones de IP's:

foreach $i ( @arrayCmd ) {    # por cada elemento $i del array @arraycmd...

    if ( $i =~ / has address / ) {   # si contiene esta cadena, tiene la IP

        $i =~ /.*\s([0-9\.]+)\s.*/;  # buscamos el patrón que contenga
                                     # solo números y puntos (IP's)
        my $ip = $1;   # asignamos el primer (y único) memorizado a $ip

         if ( $ip =~ /[0-9\.]+/ ) {   # podemos re-asegurarnos del formato IP
             print "La IP es: " . $ip . "\n";
             last;             # solo la primer secuencia de IP
         }
    }
}

exit;
```

Al ejecutar el script nos debe imprimir algo como esto:

```
user@host:# test.pl
La IP es: 74.125.45.100
user@host:#
```

Usted literalmente puede transcribir este texto a un archivo ejecutable (con permisos 755) de su equipo Servidor con SO Unix o GNU/Linux y debe trabajar perfectamente para su análisis.

A estas alturas del libro, deberíamos entender cada instrucción, no hemos usado nada que no hallamos visto.

- - -

Otro problema:
Debemos analizar los sistemas de archivos del disco duro para ver que alguno no esté por encima de un umbral que definamos de antemano.

Planteamiento:
Por medio del comando Unix *df -k* podemos visualizar la lista de sistemas de archivos del disco.

El sistema de archivos del disco duro de una PC con Ubuntu (GNU/Linux):

```
hmaza@hmaza:~$ df -k
S.ficheros         Bloques de 1K   Usado    Dispon Uso% Montado en
/dev/sda1             12389324   3682040   8077940  32% /
tmpfs                   252796         0    252796   0% /lib/init/rw
varrun                  252796       344    252452   1% /var/run
varlock                 252796         0    252796   0% /var/lock
udev                    252796      2764    250032   2% /dev
tmpfs                   252796       172    252624   1% /dev/shm
lrm                     252796      2004    250792   1% /lib/modules...sigue
/dev/sda3             25238332  17717400   6248988  74% /home
hmaza@hmaza:~$
```

Análisis:

a) - Nos interesa los porcentajes de uso, es la quinta columna. Contiene el signo de %.

b) - Obviamente, solo analizo los sistemas de archivos que yo elija:

/dev/sda3 (solo uno esta bien por ahora).

c) - Si pasa de 80% (por ejemplo) debe imprimir (no hemos aprendido mas que imprimir) algo que me indique una alarma.

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

#####################################################
#### VARIABLES DE CONFIGURACION:

# maximo porcentaje admitido sin alarmar (umbral)
$max = 80;

# sistema de archivos a monitorizar
$sisdev = "/dev/sda3";
#####################################################

$comando = `df -k`;   # cargamos las líneas del comando a una variable
```

```
@lns = split(/\n/, $comando);          # separamos cada línea y las colocamos
                                       # como elementos de un array


foreach $linea ( @lns ) {              # por cada $linea del array @lns

        if ( $linea =~ /$sisdev/ ) {                  # si contiene el sistema
                                                      # de archivos deseado...

                $linea =~ /.*\s([0-9]+)\%\s.*/;       # buscamos la columna y
                $valor = $1;                          # extraemos el valor

                if ( $valor >= $max ) {               # comparamos el valor
                                                      # con el umbral

                        print "Alarma!: $sisdev en $valor\%. Igual o por
                               encima del umbral de $max\% \n";
                }
        }
}

exit;
```

Si notamos que /dev/sda3 tiene un 74% de uso y el umbral es de 80, este script no va a imprimir nada.
Todo está bien.

Sin embargo podemos bajar el umbral para que nos muestre algo que nos haga sentir felices al ejecutar el script y que nos avise que algo anda mal.

Podemos cambiar el umbral hasta donde deseemos, en este caso, pondremos la variable **$max** con un valor de **70**, de este modo, como el sistema de archivos /dev/sda3 tiene un porcentaje de uso del 74, al ejecutar de nuevo el script, nos debe imprimir algo como esto:

```
hmaza@hmaza:~$ test.pl
Alarma!: /dev/sda3 en 74%. Igual o por encima del umbral de 70%
hmaza@hmaza:~$
```

6.2 Expresiones regulares de sustitución

Las expresiones regulares pueden servirnos para sustituir cadenas de caracteres.

La expresión básica es:

```
s/Cadena_Anterior/Cadena_Nueva/opciones;
```

Reemplaza la Cadena_Anterior por la Cadena_Nueva en una expresión.

Ejemplo:

```
$var = "Hola Mundo";
$var =~ s/Hola/Adiós/;        # sustituimos la palabra Hola por Adiós
print "$var \n";              # esto imprime: Adiós Mundo
```

Otro ejemplo, ahora con opciones:

```
$var = "Esta es una prueba";
$var =~  s/(\w+)/<$1>/gi;
print "$var  \n";             # imprime:  <Esta> <es> <una> <prueba>
```

Varias cosas que aprender:
Usamos las opciones ***gi***, las cuales significan: la ***g***, que debamos sustituir en forma recursiva, y la ***i***, que no nos interesa si son mayúsculas o minúsculas.

Otra cosa curiosa, pero que ya aprendimos, los paréntesis memorizaron la variable **$1**, la cual es usada en la sustitución, esto es, la reusamos dentro de la misma expresión.

CAPÍTULO 7
Funciones propias, subrutinas

7. Funciones propias, subrutinas

Una función propia, es llamada mas comúnmente subrutina, o solo sub, y es definida en Perl por la siguiente construcción.

```
sub nombre {
    instrucción_1;
    instrucción_2;
    instrucción_3;
}
```

Un ejemplo del famoso hola mundo:

```
#!/usr/bin/perl

sub hola {
        print "Hola mundo\n";
}
```

Si escribimos un script con solo estas líneas no hace nada, pues la subrutina no se invoca desde ningún lado.

Invocar una subrutina es "llamar" a la función propia para ejecutarla.

Básicamente es así:

```
subrutina();
```

Los paréntesis le indican a Perl que es una función propia la que se está invocando.

Sin usar los paréntesis, podemos invocar una subrutina de esta otra forma:

```
&subrutina;
```

Si al anterior script usamos un *llamado* a la subrutina...

```
#!/usr/bin/perl

hola();        # llamamos a la subrutina "hola"

exit;  # salimos del programa

sub hola {
        print "Hola mundo\n";
}
```

Ahora si debe imprimir:

```
user@host: # test.pl
Hola mundo
user@host: #
```

No importa la posición de la subrutina dentro del programa, puede incluso estar al final de todo el script, Perl sabe de ella y la ejecuta cuando uno la invoque.

7.1 La variable especial @_

Antes de proseguir, debemos hacer una escala técnica (y no me refiero a desahogar nuestras penas o deshechos corporales) para hablar acerca de una de las variables especiales de Perl.

El arreglo especial @_ está implícito siempre, dentro de cualquier script de Perl, aunque no la declaremos que existe, ésta siempre existe. Vive feliz en nuestro programa.

La variable @_ posee como todo array, elementos indizados, y estos los podemos llamar de la forma común:

```
$_[0], $_[1], $_[2], ... $_[n]
```

Un ejemplo de utilización en un bucle:

```
#!/usr/bin/perl

@array = qw(Juan Pablo Pedro);

foreach ( @array ) {
        print $_ , "\n";
}

exit;
```

No es que se nos halla olvidado escribir:

```
foreach $linea ( @array ) {
```

No necesitamos designar una variable especialmente para asignar recursivamente el valor de cada elemento, porque Perl asigna ese valor a la variable especial **$_**, la cual imprimimos.

El array @_ y sus elementos **$_[n]** estarán vacíos hasta que no las usemos como en este ejemplo que acabamos de ver.

La variable @_ siempre es local, y se destruye cuando salimos del bucle, sin embargo las variables locales y globales las estudiaremos un poquito mas adelante.

Y lo mismo sucede con los demás bucles como **while**, **for** e incluso dentro de una función propia, como veremos a continuación.

7.2 Argumentos de una función propia o subrutina

Para pasar argumentos en forma de datos a una subrutina, usamos los paréntesis de la misma sintaxis:

```
subrutina("Argumento1", "Argumento2", "Argumento3");
```

Note que los argumentos que le pasamos, forman una lista, esto es, una lista de argumentos, que puede ser en cualquier caso, un array, ya que un array es una lista de elementos:

```
subrutina( @argumentos );
```

Y además cada argumento puede ser una variable:

```
subrutina($arg1, $arg2, $arg3);
```

Así podremos usar las subrutinas pasándole los argumentos necesarios para que estas puedan usarlos para su tratamiento.

```
subrutina($carro, $moto, @animales, 32, 45);
```

En este caso Perl aplana los elementos del array y ocurre una interpolación de variables, haciendo que los elementos del array se envíen a la subrutina en el mismo orden de los índices del array, y dentro del orden de las variables de la lista de argumentos.

7.3 Variables globales y locales

Cuando "creamos" una variable en un script, esta variable existe en forma predefinida, en todo el script, y la podremos usar dentro de una subrutina o de un bucle sin problemas.

Esto es, creamos ***variables globales***.

Cuando usamos un bucle o una subrutina y deseamos que una o todas las variables no se usen fuera de la subrutina, o al salir de un bucle, podemos declarar **variables locales**.

Para esto usamos la función **my**:

Ejemplo de funcionamiento:

```
#!/usr/bin/perl

$cosa = "carro";              # $cosa es variable GLOBAL
$animal = "gato";
$numero = 45;

@array = ($cosa, $animal, $numero);

foreach my $cosa ( @array ) {        # $cosa es variable LOCAL
        print $cosa, "\n";
}                             # $cosa local se destruye al terminar el bucle

print $cosa, "\n";            # se imprime la variable global $cosa = carro

exit;
```

Esto resulta en la impresión de:

```
user@host: # test.pl
carro
gato
45
carro
user@host: #
```

Hay que hacer notar:
Declaramos una variable **$cosa** que se creó como ***variable global***.
Declaramos una variable **$cosa** para el bucle **foreach** la cual se creó como ***variable local***, mediante el uso de la función **my**.

Cuando creamos una variable local, Perl destruye esta cuando el bucle o la subrutina termina de ejecutarse.
Razón por la cual, pudimos imprimir la variable **$cosa** al final y resultó ser que tiene el valor "*carro*" que le dimos al inicio del script, y no el valor 45, que fué el último valor que se le dió en el bucle **foreach** si esta fuera variable global.

En las subrutinas :

```
#!/usr/bin/perl

hola();

sub hola {
        my $mundo = "Hola mundo";     # creamos la variable local la cual
                                      # se destruye al final de la subrutina

        print $mundo, "\n";           # imprimimos la variable local
}

print $mundo, "\n";           # no imprime mas que el salto de línea, ya que
                              # no existe la variable global $mundo

exit;
```

Esto imprime:

```
user@host: # test.pl
Hola mundo

user@host: #
```

Note que solo se imprime el salto de línea al final. Ya que la variable **$mundo** que pretendemos imprimir fué una variable local de la subrutina hola y por lo tanto fué destruida al finalizar esta.
Así, **$mundo** no existe como variable global y solo se imprime el salto de línea de la segunda función **print** que usamos.

7.4 Captando los argumentos

Ahora vamos a usar lo que acabamos de aprender para captar los valores que se le pasan a una subrutina como valores, para trabajar con ellos.

Recordemos el apartado 7.1 en donde mencionamos que:

> *La variable @_ en un bucle siempre es local, y se destruye cuando salimos de él.*

También recordemos el uso de la función **shift**, la cual le quita el primer elemento de un array y lo retorna.

Así, para quitarle el primer valor a la variable especial @_ también podemos usar **shift** para

saber de los elementos:

```
my $var = shift(@_);
```

O simplemente:

```
my $var = shift;      # Perl sabe que nos referimos a la variable especial
```

Ya que sabemos que @_ está implícita dentro del script y también dentro de cualquier bucle o subrutina como array local, y no necesitamos decir que nos referimos a @_.

Veamos un ejemplo:

```
#!/usr/bin/perl

hola("Hola mundo");

sub hola {
        my $mundo = shift;    # creamos una variable local quitando el primer
                              # elemento de la variable especial @_

        print $mundo, "\n";          # imprimimos la variable local
}

exit;
```

Igualmente, para captar los demás argumentos, si los hay, podemos usar el mismo método para tomar de uno por uno, cada elemento del array espacial @_:

```
my $uno = shift;
my $dos = shift;
my $tres = shift;
```

Igualmente podemos usar los elementos del array de la forma convencional:

```
#!/usr/bin/perl

hola("Hola mundo");

sub hola {
        my $mundo = $_[0];    # creamos una variable local llamando al primer
                              # elemento de la variable especial @_

        print $mundo, "\n";          # imprimimos la variable local
}

exit;
```

Y cada elemento del array, si hay mas de uno:

```
my $uno = $_[0];
my $dos = $_[1];
my $tres = $_[2];
```

7.5 Retornando los resultados

Una subrutina en Perl debe procesar los datos que se le pasan como argumentos y este proceso debe generar un resultado.

Este resultado puede ser devuelto por la subrutina para ser procesada por el resto del script.

Para esto se usa la función **return**.

Ejemplo:

```
sub multi {
        my $var = shift;
        $var .= " mundo";      # concatenamos algo a la variable captada

        return $var;           # regresamos el valor que queramos
}
```

De esta manera al terminar la subrutina, esta nos regresa un valor, como si se tratare una variable simple.

```
sub multi {
        my $var = shift;
        $var .= " mundo";      # concatenamos algo a la variable captada
        return $var;           # regresamos el valor que queramos
}

$result = multi("Hola");       # llamamos la función "multi" pasándole un
                               # argumento, la cadena "Hola"
                               # y tomamos lo que nos retornó la sub

print $result, "\n";           # imprime: Hola Mundo
```

7.6 Uniendo todo

Ahora veamos como unir todo para hacer las cosas mas funcionales.

Hagamos un ejemplo:

```
#!/usr/bin/perl

($hola, $num) = hola("Hola", 30);    # generamos una lista con las
                                     # variables que regresa la subrutina.
                                     # Serán variables globales

print "La cadena: " . $hola . "\n". "y el número: " . $num ."\n";

sub hola {

        my $hola = shift;      # captamos el primer argumento
        my $num = shift;       # captamos el segundo argumento

        $hola .= " mundo";     # concatenamos
        $num += 10;            # sumamos

        return ($hola, $num);        # y retornamos una lista, que al
                                     # final de cuentas es un array
}

exit;
```

Una subrutina puede devolver tanto una sola variable, o una lista de ellas, y en este caso podremos asignarlo a un array, como en el ejemplo siguiente:

```
#!/usr/bin/perl

@datos = hola("Hola", 30);    # genermos un array del retorno de la sub

print "La cadena: " . $datos[0] . "\n". "y el número: " . $datos[1] ."\n";

sub hola {

        my $hola = shift;      # captamos el primer argumento
        my $num = shift;       # captamos el segundo argumento

        $hola .= " mundo";     # concatenamos
        $num += 10;            # sumamos

        return ($hola, $num);        # y retornamos una lista, que al
                                     # final de cuentas es un array
}

exit;
```

Cuando una subrutina devuelve un único valor, este puede ser asignado a una variable escalar así:

```
$var = subrutina(40, 30, 55);

sub subrutina {
        instruccion1;
        instruccion2;

        return $valor;
}
```

Porque devolvió una variable del tipo escalar, no una lista de ellas.

7.7 Variables semilocales

Perl es capaz de generar variables locales que puedan ser usadas solamente dentro de esa subrutina y dentro de algunas subrutinas que sean colocadas dentro esa subrutina inicial.

Lo explicamos, si creamos una subrutina, y dentro de esta creamos otra subrutina, podremos declarar una variable local para la subrutina externa, pero global para la o las subrutinas internas.

Ejemplo:

```
#!/usr/bin/perl

$value = "GLOBAL";

simple();
doble();
simple();

sub doble {
    local $value = "LOCAL";          # creamos una variable semilocal
                                     # que puede ser usada dentro de otra
                                     # subrutina como si fuera global
                                     # solamente dentro de la subrutina
                                     # externa

    simple();                        # se imprime el valor local
}

sub simple {
                # se imprime el valor global
    print "El valor actual de \$value es $value\n";
}

exit;
```

Corremos el script y debe imprimir en pantalla:

```
user@host:~$ test.pl
El valor actual de $value es GLOBAL
El valor actual de $value es LOCAL
El valor actual de $value es GLOBAL
user@host:~$
```

7.7.1 Reutilización de código

De esta forma podemos crear funciones propias, llamadas comúnmente subrutinas, las cuales podremos reusar en distintas etapas de un script de Perl.

Esto se llama *reutilización de código* y se usa normalmente para ahorrar precisamente código, tiempo y carga en el servidor, que al final del día se traduce como ahorro en efectivo.

En muchas ocasiones podremos reusar estas subrutinas para diferentes scripts que no sean para los cuales fueron escritos, y esto lo podemos hacer fácilmente generando un archivo de subrutinas y "llamando" este archivo desde el script de Perl actual para integrarlo como si fuera parte de él.

Lo anterior se logra mediante el uso de la función **require**:

Sintaxis:

```
require "path/to/file";
```

require inserta el código dentro del archivo, en el espacio correcto dentro del script de Perl.

Por ejemplo, si hacemos una subrutina y la colocamos en el archivo subs.pl (usamos este nombre para identificarla del script ejecutable, usted puede ponerle el nombre que desee).

Y posteriormente creamos el script de Perl, que llame al archivo con la(s) subrutina(s).

Ejemplo:

Vamos a crear un mecanismo para generar password aleatorio.

Archivo **subs.pl**:

```
##### archivo subs.pl que guarda subrutinas para diferentes usos

sub genera_passwd {
        my $max = shift;
        my @chars = (A..Z, a..z, 0..9);      # generamos rangos de caracteres
                                             # de letras y números que
                                             # puedan usarse en un password
        my $contra;

        for ( 1 .. $max ) {
                $contra .= $chars[rand @chars];      # usando la función rand
                                             # seleccionamos aleatoriamente
                                             # un elemento del array
        }

        return $contra;
}

1;      # al retornar al script debe ser con valor verdadero
```

Lo que hacemos aquí es seleccionar un índice del array @array para concatenarlo a mas elementos seleccionados y obtener una mezcla de caracteres aleatorios que formen un password.

Dos cosas que aprender mas.

a) Aquí usamos una función mas de Perl, **rand**, la cual genera en forma aleatoria un número de los que le pasemos como argumento.

b) Al final del archivo que insertamos con **require**, colocamos una línea: ***1;***, la cual nos asegura que la inserción mediante **require** se terminó correctamente y entonces regresa un valor de verdadero, lo cual nos permite continuar con el script en forma correcta.

Note además, que no es un script de Perl propiamente dicho, ya que no se ejecuta. Y debido a esto, desde luego, no es necesario que tenga permisos de ejecución.

Ahora el script de Perl:

```
#!/usr/bin/perl

my $maxChar = 6;              # configuramos el largo del password

require "subs.pl";            # llamamos e insertamos el código dentro del
                              # archivo subs.pl en esta parte del script


my $passwd = genera_passwd($maxChar);  # corremos la subrutina que
                                       # físicamente se encuentra en el
                                       # archivo subs.pl, pero que ahora
                                       # forma parte del script
print "Tu password es: $passwd \n";

exit;
```

Ejecutamos el script:

```
user@host:~$ test.pl
Tu password es: 1EwSS1
user@host:~$
user@host:~$ test.pl
Tu password es: KdFLT6
user@host:~$
user@host:~$ test.pl
Tu password es: DiLOl3
user@host:~$
user@host:~$ test.pl
Tu password es: 7Q5ub0
user@host:~$
```

Cada vez que ejecutamos el script, nos imprime una cadena diferente.

A estas alturas todo lo que hacemos en este script debe ser comprendido... o eso espero.

CAPÍTULO 8
Entrada/Salida en Perl. Archivos

8. Entrada/Salida en Perl. Archivos

Perl interactúa con el usuario o con el sistema operativo por medio de Entradas/Salidas (Input/Output o simplemente I/O) que permiten el intercambio de información.
Este intercambio de datos se gestiona por medio de manejadores (handle) específicos que mantienen abierta una interfaz entre el script y su entorno.

8.1 Manejadores de entrada/salida de datos (I/O Filehandle).

Un Filehandle en Perl, es el nombre de una conexión entre el proceso de Perl y el mundo exterior.

De forma implícita (como la variable @_ que estudiamos anteriormente) Perl crea algunos manejadores de **I/O** que no necesitamos declarar cuando los usamos.

De estos manejadores ya hemos usado algunos sin darnos cuenta a lo largo del estudio de este libro:

STDOUT (Standard Output) La salida estandard o bien, nuestra terminal o monitor.

STDIN (Standard Input) La entrada estandard o bien, el teclado.

STDERR (Standard Error output) Es la salida de errores del sistema.

8.2 La entrada y la salida estandard

Podemos ver ahora que al usar la función **print**, solo le hemos dicho qué imprimir y Perl interpreta hacia que manejador va a dirigir la salida, y ahora nos damos cuenta que es la STDOUT.

```
print "Algo que imprimir\n";
```

Es lo mismo que:

```
print STDOUT "Algo que imprimir\n";
```

Aunque no necesitamos declarar que vamos a dirigir la salida a la Standard Output.

Perl acepta una nomenclatura sin un prefijo especial (como en las etiquetas) para los manejadores I/O.

Se recomienda usar siempre una nomenclatura con MAYÚSCULAS para evitar confusiones durante el desarrollo del script.

Perl también nos puede permitir usar una variable escalar como manejador, por ejemplo $miarchivo para declarar un manejador, el cual debe estar dedicado solo para este uso.

Así, podemos usar *miarchivo*, o *MIARCHIVO*, o bien, *$miarchivo*, de los que se recomienda, como ya se dijo, *MIARCHIVO*, en mayúsculas.

- - -

Ahora, podemos también administrar la entrada estandard STDIN:

```
#!/usr/bin/perl

print "Cual es tu nombre? ";

$name = (<STDIN>);     # El script espera a que ingreses una cadena y enter
                       # si no, no puede continuar

chomp $name;           # quitamos el salto de línea, si lo hay

print STDOUT "Bienvenido $name\n";   # STDOUT sale sobrando

exit;
```

8.3 Abrir y cerrar archivos en Perl

Existe una función de Perl para declarar filehandles adicionales, es la función ***open***, la cual tiene la siguiente sintaxis para la apertura de archivos.

Sintaxis para abrir un archivo:

```
open (MANEJADOR, "modo_de_acceso + nombre/de/archivo")
```

Sintaxis para cerrar el mismo archivo:

```
close (MANEJADOR);
```

El nombre del archivo incluye su ruta de acceso.

Los modos de acceso mas básicos son:

```
>       Escritura. Vacía el archivo antes de escribir. Ojo con este.
>>      Escritura. Escribe al final del archivo, concatenando el contenido.
<       Lectura.   Si no se establece modo_de_acceso, este es por defecto,
                   y abre un archivo para su lectura.
```

8.4 El acceso a archivos

Una vez que abrimos un archivo para lectura, lo podremos accesar por medio de los operadores de datos **<** y **>**, los cuales "leen" el archivo línea por línea.

Ejemplo:

Supongamos que queremos leer las líneas del archivo /etc/passwd de nuestro sistema operativo Unix o GNU/Linux.

```
open (CONTRAS, "/etc/passwd");       # abrimos el archivo para lectura
                                     # también podemos usar "</etc/passwd"

while (<CONTRAS>) {                  # leemos línea por línea

    chomp;                    # quitamos los saltos de línea de $_

    print "$_ \n";            # imprimimos la línea en turno
}
close CONTRAS;
```

También podemos cargar todo el contenido del archivo en un array. En este caso Perl identifica que cada salto de línea representa el límite de los elementos, poniendo cada línea como un elemento del array:

```
open (IN, "</etc/passwd");    # abrimos el archivo para lectura

my @lineas = <IN>;            # cargamos todo el archivo en un array

close IN;
```

Este último método tiene la desventaja de que si el archivo es muy grande, la memoria de nuestro equipo se ve disminuida, porque cargamos todas las líneas del archivo en una sola variable.

Igualmente, podemos imprimir en un archivo, según el modo de apertura:

```
open (OUT, ">/tmp/file.txt");        # abrimos el archivo para escritura
                                     # vaciándolo antes

print OUT "Esta es una línea de prueba\n";  # se imprime en el archivo
                                            # debido al manejador declarado
close OUT;     # cerramos el manejador
```

8.5 Manejo de errores para I/O de archivos

Al comanzar a trabajar con archivos, en muchas ocasiones tendremos problemas para abrir archivos por distintas razones.

El caso ahora es aprender a manejar esos errores a nuestra conveniencia.

8.5.1 Algunos métodos simples

Si recordamos la función unless, nos ejecuta un bloque cuando la expresión evaluada nos devuelve falso:

```
my $datafile = "/tmp/file.txt";

unless ( open (DATA, ">$datafile") ) {
    print "No se puede crear $datafile\n";
} else {
    my @arr = <DATA>;

    # etc, etc...
    # ... y el resto del programa se ejecuta completo cuando
    # la expresión es verdadera, o sea, si se abre bien el archivo.
}
```

Claramente vemos que si no se puede abrir el archivo, no se ejecuta el resto del programa, el cual termina sin hacer nada.

8.5.2 La salida die

Otro método un poco mas fácil, eliminando el complemento de ***unless***, ***else***, es usar la función ***die***, la cual termina un script cuando la usamos:

```
my $datafile = "/tmp/file.txt";

unless ( open (DATA,">$datafile") ) {
```

```
    die "No se puede crear $datafile\n";
}
my @arr = <DATA>;
# etc, etc...
# ... y el resto del programa se ejecuta completo cuando
# la expresión es verdadera, o sea, si se abre bien el archivo.
```

Aquí vemos que no necesitamos else para correr el resto del programa cuando la expresión evaluada resulta verdadero, ya que si resulta falso, simplemente salimos del programa.

Adicionalmente vemos que podemos imprimir algún mensaje de error cuando esto sucede.

8.5.3 El operador *or* y el operador ||

Podemos simplificar mas las cosas usando el operador ***or***, en caso de falla al abrir un archivo:

```
open (DATA,">$datafile") or die "No se puede crear $datafile\n";
```

Usualmente podemos trabajar mas fácilmente con el operador ||:

```
open (DATA,">$datafile") || die "No se puede abrir $datafile\n";
```

Tanto para lectura o escritura, todo funciona igual.

Existe otra variable especial en Perl que nos devuelve la cadena del mensaje de error, y es la variable **$!**, la cual podemos usar así:

```
open (DATA,">>/tmp/file.log") || die "Error: $!\n";
```

8.5.4 Diferentes salidas

Pueden existir ocaciones en que no queramos salir del script, sino administrar manualmente los errores.

En este caso podemos olvidarnos de la función ***die*** y usar por ejemplo, una desviación hacia una subrutina.

Regresemos al ejemplo de las alarmas de sistemas de archivos visto en el punto 6.1.1.

Ahora en lugar de imprimir la alarma en la pantalla, queremos imprimirla en un archivo que en este caso será /mnt/remoto/alm.txt, notando que talvez sea un directorio montado remotamente, y es probable que en algún momento la conexión no esté disponible, por lo que deseamos guardar el error en otro archivo local... por si las dudas.

```
#!/usr/bin/perl

$max = 70;                              # umbral de alarma
$sisdev = "/dev/sda3";                  # sistema de archivos a monitorizar
$almfile = "/mnt/remoto/alm.txt";       # archivo de alarmas
$logfile = "/tmp/test.log";             # archivo log

$comando = `df -k`;   # cargamos las líneas del comando a una variable

@lns = split(/\n/, $comando);           # separamos cada línea y las colocamos
                                        # como elementos de un array
chomp @lns;

foreach $linea ( @lns ) {               # por cada $linea del array @lns ...

        if ( $linea =~ /$sisdev/ ) {                # si contiene el sistema
                                                    # de archivos deseado...

                $linea =~ /.*\s([0-9]+)\%\s.*/;     # buscamos la columna y
                $valor = $1;                        # extraemos el valor

                if ( $valor >= $max ) {             # comparamos el valor
                                                    # con el umbral
                        # (en una sola línea)
                    $msg = "Alarma!: $sisdev en $valor\%. Igual o por  encima
                    del umbral de $max\%";

                        # aquí llamamos una subrutina si falla la función open
                    open (ALM, ">>$almfile") || sublog($almfile, $msg);
                    print ALM "$msg\n";
                    close ALM;
                }
        }
}

exit;

sub sublog {
        my $file = shift;
        my $msgAlm = shift;
        my $msglog = "No se pudo abrir $file. Mensaje fallido: '$msgAlm'";

        open (LOG, ">>$logfile") || die;     # ya no usamos otra opción
        print LOG "$msglog\n";
        close LOG;
        exit;           # ahora si, forzamos la salida
}
```

Si revisamos nuestro sistema de archivos monitorizado /dev/sda3 y colocamos el umbral por debajo de lo que tengamos, el script debe buscar la ruta del archivo /mnt/remoto/alm.txt, el cual como en realidad no existe (no lo hemos creado para la prueba) nos debe llevar a la ejecución de la subrutina *sublog*, la cual imprime en el archivo /tmp/test.log .

Al abrir el archivo /tmp/test.log contendrá una sola línea parecida a esto:

```
No se pudo abrir /mnt/remoto/alm.txt. Mensaje fallido: 'Alarma!: /dev/sda3 en
74%. Igual o por encima del umbral de 70%'
```

8.6 La seguridad en el manejo de archivos

De nada sirve aprender el mejor lenguaje de programación si no se tiene una seguridad de archivos en buen estado, sin corrupción o perdidos.

Las siguientes líneas están dedicadas a ello, y debe usted poner toda su fina atención en este tema.

Lo mas importante al manejar datos, son precisamente unos datos completos.

8.6.1 File Test, operadores de comprobación de archivos

Que tal si queremos manejar un archivo de nombre salida.log para administrar algunas salida de errores de un script, pero (siempre hay un pero) no sabemos si alguna vez ya creamos ese archivo con otro script y entonces mezclaremos los datos ya existentes con los nuevos generando un archivo con líneas de diferentes formatos para analizar.

Entonces generaremos un caos.
Para evitar esto, podemos usar los file test de Perl que nos dan información acerca del archivo que estamos manejando.

Ejemplo usando -e (devuelve verdadero si existe):

```
$archivo = "salida.log";
if (-e $archivo) {
        print "Hey! el archivo $archivo ya existe! busca otro nombre\n";
} else {
        print "El archivo no existe y se va a crear...\n";
        # y sigue el programa
}
```

Si el archivo existe, la expresión con ***-e*** retorna verdadero, en caso contrario devuelve falso. Hay muchos file tests, y podemos ver algunos a continuación:

```
-r      Archivo o directorio es de lectura

-w      Archivo o directorio es de escritura

-x      Archivo o directorio es ejecutable

-o      Archivo o directorio es propio del usuario

-R      Archivo o directorio es de lectura para el usuario real

-W      Archivo o directorio es de escritura para el usuario real

-X      Archivo o directorio es ejecutable para el usuario real

-O      Archivo o directorio es propio del usuario real

-e      Archivo o directorio existe

-z      Archivo existe y tiene tamaño cero o vacío.

-s      Archivo o directorio existe y tiene tamaño diferente de cero
(el valor retornado es el tamaño en bytes)

-f      Es un archivo plano

-d      Es un directorio

-l      Es un symlink

-S      Es un socket

-p      Es un pipe declarado (un "fifo")

-b      Es un archivo block-special (como un disco montado)

-c      Es un archivo character-special (como un I/O device)

-u      Archivo o directorio es setuid

-g      Archivo o directorio es setgid

-T      Archivo es de tipo "text"

-B      Archivo es de tipo "binary"

-M      Última modificación en días

-A      Último acceso en días

-C      Última modificación del inodo en días
```

El file test ***-s*** devuelve el tamaño en bytes del archivo y su uso puede ser:

```
my $filesize = -s "/var/log/esearchivo.log";

print "El tamaño del archivo es $filesize bytes\n";
```

8.6.2 Bloqueo de archivos

En muchas ocasiones es posible que un mismo archivo esté siendo usado por mas de un proceso o script de Perl, y es fácil llegar a tener errores en esos archivos.

Para evitar estas catástrofes y llenarnos de archivos inutilizables y perder nuestro trabajo, es muy aconsejable tener siempre mecanismos de seguridad cuando abrimos un archivo, evitando que otros puedan hacer uso del mismo, si nuestro script ya lo abrió primero.

Se usa de manera muy común el bloqueo de los archivos, tanto para escritura como para lectura.
De esta forma, solo un proceso puede hacer uso del mismo a un tiempo.

Haremos un paréntesis para explicar brevemente algo muy importante en Perl, y que debemos saber para las siguientes líneas.

Perl posee una función de nombre ***flock()*** la cual hace un llamado a la función ***flock*** del sistema operativo, o bien, una emulación de esta.

Sintaxis de flock():

```
flock(FILEHANDLE, MODO);
```

Donde FILEHANDLE es el identificador del archivo para abrir, y MODO es la forma en que se está abriendo.

Los modos mas comunes que usamos son, solo lectura (**1**) y completo (**2**).

Ejemplos:

```
open (FILE, "<mensajes.txt");              # abrimos para lectura

flock (FILE, 1);             # bloqueamos el archivo para que solo puedan
                             # leerlo cuando nosotros lo usamos
my @mensajes = <FILE>;
close FILE;
```

De esta forma, nuestro script lo abre, lo bloquea y lo lee, dejando que otros procesos solo pueden leer el contenido, no modificarlo.

```
open (FILE, ">>mensajes.txt");       # abrimos para escritura

flock (FILE, 2);              # bloqueamos el archivo por completo para que
                              # no pueda ser accesado mientras lo usamos
print FILE "Esta es una linea insertada\n";
close FILE;
```

Cabe mencionar que la función flock trabaja correctamente en casi todos los sistemas operativos, excepto los que no poseen esta función a nivel OS, por ejemplo, MSWindows©.

Además de la función **flock()**, existen diversos **módulos** para proteger el uso de archivos, veremos uno que otro, no sin antes platicar sobre los mencionados **módulos**.

8.6.2.1 Muy breve introducción al uso de módulos

Si antes hablamos de subrutinas que pueden estar en otros archivos no ejecutables, los cuales podían ser llamados mediante la función **require**, ahora podemos hacer uso de otro tipo de archivos que también llevan varias funciones dentro, aunque su construcción es especialmente diferente de un archivo de subrutinas común.

Son los módulos de Perl.

Estos deben llevar, como se mencionó, una construcción especial del que no es el caso hablar de ello, sin embargo sí debemos hablar de cómo se llaman los módulos desde un script.

Un módulo en Perl siempre debe terminar en **.pm**, lo cual lo identifica como módulo de Perl.

Aunque lleve una construcción diferente, es un grupo de subrutinas, y es posible llamarlo con la función normal require.

Ejemplo:

```
require "modulos/modulo.pm";
```

Y podremos usar todas las funciones dentro de éste. (Note que es un módulo dentro de un directorio)

El método mas exacto para llamar un módulo es mediante la función **use**.

Ejemplo:

```
use modulos::modulo;
```

Note que no le colocamos la terminación **.pm**.

Note además, que *modulos* es un sub-directorio que contiene el archivo *modulos.pm*, y este se separa mediante un doble caracter : (**::**)

Con este método además, podemos importar las constantes específicas para nuestro uso en el script, y para no cargar todas las funciones.

Perl, en sus distribuciones estandard, viene con una buena cantidad de módulos que fueron escritos por personas que les interesa que todos los podamos usar, y que nos resuelven muchos problemas.

Podemos así, reutilizar código de otros programadores.

8.6.2.2 Uso de flock del módulo Fcntl

La función **flock()** que emula el **flock** del Sistema Operativo tiene algunos problemas de uso especialmente con algunos sistemas operativos.

Para evitar esto, podemos trabajar con bloqueos mediante el módulo genérico **fcntl**, del cual podemos importar las constantes **LOCK_***.

Pero veamos un ejemplo:

```
#!/usr/bin/perl

use Fcntl ':flock';   # importamos el método flock que incluyen
                      # las constantes LOCK_* del módulo Fcntl

open (FILE, "</path/mensajes.txt");

flock(FILE, LOCK_SH);         # LOCK_SH... de shared

my @mensajes = <FILE>;

close FILE;
```

Aquí bloqueamos el archivo para lectura con LOCK_SH y queda compartido para que pueda ser leído por otros procesos.

Y en este ejemplo:

```
#!/usr/bin/perl

use Fcntl ':flock';   # importamos el método flock
                      # y las constantes LOCK_* del módulo Fcntl

open (FILE, ">>/path/to/file");

flock(FILE, LOCK_EX);        # LOCK_EX... de exclusive

print FILE "Este es un mensaje en una linea\n";

close FILE;
```

Bloqueamos el archivo en forma total con LOCK_EX, para que no pueda ser accesado ni para lectura y menos para escritura.

Este método es limpio y no deja rastros de uso, como la función simple flock().

- - -

8.6.3 Método Maza de seguridad (del autor)

Que tal si queremos abrir un archivo para escribirlo y nuestro script se da cuenta de que no lo puede abrir, debido a que otro script lo está usando, en ese caso y de acuerdo a un script anterior, de la sección 8.5.4, nos escribirá en un archivo log una línea que nos indica que no pudo abrir el archivo para escribirlo.

Si no queremos que eso pase y deseamos que intente en forma repetida el abrir el archivo hasta que lo consiga (cuando el otro proceso lo desbloquee), entonces podemos hacer uso de nuestra pequeña experiencia y usar algún método de reintento.

Su servidor, el autor de este libro, o sea yo, hice una pequeña rutina para esperar a que otro proceso "suelte" el archivo y en ese entonces, tomarlo para trabajar en él.

Es muy simple, pero funciona.

Código del método Maza de seguridad para archivos:

```
while (1) {    # generamos un bucle infinito

    flock(FILE, LOCK_EX) or next;    # si no se puede bloquear, sigue con
                                     # el ciclo ...y vuelve a intentarlo

    last;      # cuando llegue a este punto es porque sí se pudo bloquear
               # y entonces sale del bucle while para no intentar de nuevo
}
# ...y continuamos con el script
```

De esta forma nos aseguramos de que:

El archivo a usar deba estar libre para su uso
El archivo lo bloqueamos cuando deje de estar ocupado
El script no prosigue hasta que se pueda bloquear el archivo. Tampoco termina.

Adicionalmente podemos ponerle un tope al bucle y esperar solo *n* intentos de bloqueo, en lugar de hacerlo infinito.
O talvez podamos temporizarlo para esperar unos minutos y después, si no conseguimos bloquearlo proceder a guardar el error en el archivo log.

Si reducimos el método a su mínima expresión:

```
# metodo Maza de seguridad

while (1) { flock(FILE, LOCK_SH) or next; last; }
```

Ahora hagamos el mismo script, con reintentos y con protección de intentos:

```
$cnt = 0;              # inicializamos el contador de protección
while (1) {            # generamos un bucle infinito
       $cnt ++;        # incrementamos en 1 cada vez

               # si llega al límite $maxEspera, a hora si, la subrutina
    sublog( "/path/to/file", $msg ) if $cnt eq $maxEspera;

    flock(FILE, LOCK_SH) or next;
    last;
}

print FILE "$msg\n";
close FILE;
```

Y si queremos asegurarnos de que se abra, además de que se bloquee correctamente?

El ejemplo completo quedaría así:

```
#!/usr/bin/perl

use Fcntl ':flock';
$logfile = "/tmp/test.log";          # archivo log
$maxEspera = 100000;                 # máximo número de intentos
$msg = "Este es un mensaje en una linea";

# método Maza de seguridad, aplicado a la función open
$cnt = 0;
while (1) {
    $cnt ++;
    sublog( $logfile, "Fallo en apertura" ) if $cnt eq $maxEspera;

    open(FILE, ">>/path/to/file") or next;
    last;
}

# método Maza de seguridad, aplicado al bloqueo de archivos
$cnt1 = 0;
while (1) {
    $cnt1 ++;
    sublog( $logfile, "Fallo en bloqueo" ) if $cnt1 eq $maxEspera;

    flock(FILE, LOCK_SH) or next;
    last;
}

print FILE "$msg\n";
close FILE;

exit;

sub sublog {
        my $file = shift;
        my $msgAlm = shift;
        my $msglog = "No se pudo abrir $file. Mensaje de falla: '$msgAlm'";

        open (LOG, ">>$logfile") || die;     # ya no usamos otra opción
        print LOG "$msglog\n";
        close LOG;
        exit(1);              # y ahora si, forzamos la salida (sin error)
}
```

De este modo, si no conseguimos abrir un archivo después de un número de intentos determinado, generamos una línea en el archivo log.

Del mismo modo, si no podemos bloquear un archivo después de un determinado número de intentos, también generamos una línea en el archivo log, con el debido mensaje de error.

Con esto podemos ya comenzar a trabajar en forma segura con archivos.

Queda pendiente el uso del método con temporizador en lugar de contador de intentos.

Si a usted se le ocurre otro método simple y fácil para asegurar el manejo de los datos en archivos, no dude en compartirlo con el resto del mundo, hay que regresar un poco de lo mucho que éste nos ha dado.
Tal vez quiera enviarlo a mi correo y en la próxima edición lo publicamos con su debido mensaje de autoría.

8.7 Manipulación de archivos y directorios

Perl posee funciones parecidas a las funciones de los Sistemas Operativos Unix o GNU/Linux, los cuales tienen usos similares.

Además de la manipulación de archivos, incluimos también la de directorios, pues es algo que siempre debemos hacer en nuestro caminar por el mundo de Servidores.

8.7.1 Removiendo un archivo

La función **unlink()** borra un archivo.

Sintaxis:

```
unlink ("file");
```

Como ejemplo, hagamos un comando para borrar un archivo:

```
#!/usr/bin/perl

$aborrar = shift;      # captamos el argumento $_[0]

unlink $aborrar;       # borramos el archivo dado por el argumento

exit;
```

Si lo ejecutamos:

```
hmaza@hmaza:~$ del.pl /tmp/test.log
hmaza@hmaza:~$
```

Esto nos borra el archivo que estábamos usando como salida de log de errores que le pasamos como argumento y claro, nos deja sin poder analizarlos.

8.7.2 Renombrando un archivo

Para renombrar un archivo usamos la función **rename()**.

Sintaxis con ejemplo:

```
rename("nombre_antiguo.pl", "nombre_nuevo.pl");
```

8.7.3 Creando y removiendo directorios

Las funciones **mkdir()** y **rmdir()** son capaces de crear y remover directorios,claro, vacíos.

Sintaxis con ejemplo:

```
mkdir("directorio", 0777) ;
```

Crea un directorio con los permisos que se desee.

```
rmdir("directorio");
```

Remueve un directorio si éste está vacío.

8.7.4 Permisos

En forma similar al comando *chmod* de Unix, tenemos la función **chmod()** de Perl.

Sintaxis con ejemplo:

```
chmod(0666, "archivo");
```

o bien, modificando varios archivos a la vez:

```
chmod(0666, "file", "archivo", "otro_archivo");
```

8.8 Otros modos de acceso

Además de los modos de acceso a archivos que ya vimos, como el **>>** o **<**, podemos acceder a un archivo no para abrirlo a lectura o escritura, sino para ejecutarlo y pasarle

argumentos, recibir datos, etc.

8.8.1 Los canales PIPEs de comunicación:

Para esto usamos en lugar de los modos de acceso de lectura o escritura, el pipe (|) para informarle a la función **open**() que queremos establecer un canal de comunicación con un archivo ejecutable.

Para ilustrar el uso del pipe, generaremos un script que envíe un correo electrónico mediante el comando de Unix sendmail, el cual debemos saber, la sintaxis básica es:

```
user@host:~$ sendmail -t...
```

Veamos el ejemplo:

```
#usr/bin/perl

my $sendmail = "/usr/sbin/sendmail -t";
my $from = "sender\@dominio.com"; # no olvidemos el backslash
my $to = 'destinatario@dominio.com'; # recordemos las comillas simples
my $subject = "Mensaje de prueba";

open(MAIL, "|$sendmail");            # abrimos el comando para ejecutarlo

print MAIL "From: $from\n";          # el From: de sendmail
print MAIL "To: $to\n";              # el To: de sendmail
print MAIL "Subject: $subject\n\n";         # Subject:, de sendmail
print MAIL "Este es un mensaje de prueba, para ver si funciona\n\n";
print MAIL "Si lo recibe, favor de no hacer caso\n";

close MAIL;                  # cerramos el canal de comunicación
exit;
```

Obviamente, el programa sendmail debe estar instalado en el servidor para poder usarlo con Perl.

En este ejemplo vemos que la función **open()**, abre un canal de comunicación con ***sendmail***, y posteriormente imprimimos varios comandos como el conocido *From:*, y otros mas hasta cerrarlo y enviar de este modo, el correo electrónico.

Esto lo podemos usar para notificar las alarmas de los servidores cuando hay problemas.

Lo veremos en las siguientes páginas.

CAPÍTULO 9
Desarrollo de scripts

9. Desarrollo de scripts

Ahora podemos comenzar a desarrollar programas que hacen algo por nuestros Servidores o los procesos que mantienen trabajando.

Es el caso mencionar que podemos usar o mejor dicho, reusar código que ya ha sido escrito y comprobado que funciona adecuadamente.

Ya aprendimos ligeramente a usar los módulos de Perl, que en la realidad pueden ser Paquetes.
Aunque pueden ser lo mismo, hay diferencias entre ambos.

Un Paquete en Perl, tiene provisto una protección espacial dentro del script, es una sección de código que puede o no, interactuar con el resto del script o con otros Paquetes.

Un paquete puede estar colocado dentro de un script, delimitado solo por el uso de la declaratoria de Paquete:

```
package NombrePaquete;
# resto del paquete
```

Un Módulo es un Paquete reusable que ha sido definido como librería para su uso.

Un módulo es pues, un archivo o una librería que contiene un Paquete o colección de ellos.

El archivo del Módulo debe terminar forzosamene en **.pm** (ya lo habíamos visto).

Ya aprendimos que llamamos a un módulo dentro de un script por la función ***use***, prefiriéndola sobre el uso de ***require*** por razónes de seguridad e importación de constantes y funciones.

Ahora veremos algunos módulos y la forma en que nos pueden ayudar a crear scripts de uso en la empresa con grandes o muy grandes (o incluso con pequeñas) capacidades de manejo datos.

Repetimos que no es la intención de este libro profundizar con Módulos y Paquetes ni su construcción, por lo que solo nos limitaremos a su explotación para nuestros propósitos.

9.1 El pragma strict

Los pragmas (que son muchos. Ver: http://perldoc.perl.org/index-pragmas.html) son directivas del compilador y con esto le indicamos como debe actuar con nuestros scripts.

Strict es un pragma para restringir la seguridad de nuestras construcciones.

Llamamos al pragma strict como se llama un módulo común:

```
        use strict;
o
        use strict "vars";    # solo restringe variables
o
        use strict "refs";    # solo restringe referencias simbólicas
o
        use strict "subs";    # solo restringe las subrutinas
```

Si no declaramos una lista a importar, se asumen todas las restricciones posibles

En la práctica es ampliamente usada la forma general de strict por ser la mas segura:

```
        #!/usr/bin/perl
        use strict;
```

Cuando usamos el pragma strict habrá que declarar o inicializar cada una de las variables que usemos, así como las que estén dentro de las funciones de Perl y las propias, ya sea con ***my***, ***our***, ***local*** o cualquier método del mismo efecto.
También habrá que entrecomillar las variables cuando no son subrutinas o identificadores de archivo.

Esto nos ayuda a formalizar cada una de las rutinas, bifurcaciones, bucles y salidas de nuestro script, haciendo este mas seguro y correcto en su funcionamiento y desarrollo, y eliminando las famosas inconsistencias que nos pueden dar muchos dolores de cabeza.

En resumen, hay que usarlo siempre.

9.2 Objetos de los módulos

Que es un objeto?

Mucha debe ser la curiosidad del lector para saber que es un objeto, como funcionan y

como usarlo, pues se oye continuamente de estos en muchos scripts.

En nuestra obra solo trataremos lo último mencionado, cómo usarlos, pues no es la obra dedicada a aprender la construcción de objetos en Perl, cosa para la cual existen documentos especializados.

> *Un objeto es una variable, siempre referenciada, que engloba en su interior datos y código, y que sabe siempre a que clase pertenece.*
>
> *Una clase es simplemente un paquete que sabe exactamente que métodos proporcionar para tratar los objetos.*
>
> *Un método es simplemente una subrutina que espera una referencia objetiva (o un nombre de paquete, para métodos de clase) como su primer argumento.*

Fácil no ??

Para poder hacer uso de un paquete o módulo debemos forzosamente ir al repositorio de éstos, en donde encontraremos los métodos (o subrutinas) que podamos usar y cómo debamos usarlos.

La URL del repositorio mas grande de Perl, el CPAN, es:

```
http://search.cpan.org/
```

En donde se tiene un buscador por medio del cual podremos encontrar mediante una palabra clave, algo que nos ayude a trabajar con nuestro proyecto.
Casi siempre hay algo que nos apoye.

Y desde luego, junto con cada módulo para Perl, habrá siempre explicaciones y ejemplos para uso de los métodos que poseen.

9.3 Algunos Ejemplos prácticos

Uno de los objetivos de este libro es aprender algunas cosas que nos ayuden a automatizar tareas, o a monitorizar recursos o procesos.

En esta parte de la obra pretendemos colocar una pequeña parte de nuestra experiencia en el ambiente de Servidores en el rubro antes mencionado.

Los siguientes ejercicios ayudarán al lector a obtener un acercamiento a los scripts de Perl y a perderle el miedo a éstos.

Póngase cómodo porque daremos un pequeño paseo por el fabuloso mundo de los Servidores industriales.

Hay que iniciar con algo simple...

9.3.1 Ejercicio 1:
Extraer la fecha del sistema para manejarla en un script.

Primero debemos saber de la existencia de dos funciones mas de Perl que pueden usarse en un script de Perl:

time Retorna el número de segundos desde el 1 de Enero de 1970

localtime Convierte una cadena en segundos y regresa una lista de 9 elementos: ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst)

Combinando estas dos funciones nos retorna datos importantes de la fecha y hora local:

```
localtime(time);
```

Sabremos los segundos, minutos, etc. y el año en el que estamos actualmente con la simple orden:

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime(time);
```

O bien:

```
my @fecha = localtime(time);
```

Conociendo el orden de los elementos sabremos cuales son e imprimir los datos:

```
#!usr/bin/perl
@fecha = localtime(time);

foreach (@fecha) {
        print $_, "\n";
}
```

Que imprimirá los nueve elementos devueltos por localtime.

Pero al año nos lo mostrará como 109 (si estamos en el 2009), pero como sabemos que nos muestra la cantidad de años transcurridos desde 1900, así que debemos agregarle al elemento que representa el año, el 5° elemento (como la película), 1900 mas, así:

```
#!usr/bin/perl
@fecha = localtime(time);

$fecha[5] += 1900;    # es el elemento que representa el año

foreach (@fecha) {
        print $_, "\n";
}
```

De esta manera, el sexto elemento nos mostrará 2009 en lugar de 109.

Recordemos la función **map**, la cual recorre iterativamente los elementos de un array o lista, y puede cambiarlos si lo deseamos.

La sintaxis:

```
map { INSTRUCCIONES_POR_CADA_ELEMENTO } @array;
```

El mismo script anterior lo podemos hacer así:

```
#!/usr/bin/perl
@fecha = localtime(time);

$fecha[5] += 1900;

map { print $_, "\n" } @fecha;
```

Ahora, el mes obviamente estará equivocado, ya que la función **localtime** nos regresa desde el mes cero hasta el 11 (que son 12 meses del año en total), y debemos agregarle 1 al elemento del mes:

```
$fecha[4] ++;
```

Además, como los segundos, minutos, etc. que se nos devuelve con **localtime** no tienen el formato que deseamos tener, por ejemplo, 02 o 05, debemos agregarle un cero a las constantes que son menores a 10.

Para esto debemos "transformar" el contexto numérico a cadena de texto de ese elemento.

Con la función **map** haremos esto:

```
@fecha = map {
        if ($_ < 10) {          # Si es menor a 10
                $_ = "0$_";     # agregamos un cero a la izquierda y
                                # lo manejamos como cadena de texto
        } else {
                $_;             # si no, lo dejamos así... Esto es relleno
        }
} @fecha;
```

Ahora, si deseamos que este método esté disponible para ser reutilizado por otras subrutinas durante todo el script, la convertimos en función propia:

```
#!/usr/bin/perl
sub sysdate {                   # nuestra función
        @f = localtime(time);
        $f[5] += 1900;
        $f[4] ++;               # el mes 0 será el mes 1 y así....

        @f = map {
                if ($_ < 10) {          # Si es menor a 10
                        $_ = "0$_";     # agregamos un cero a la izquierda
                }else{
                        $_;
                }
        } @f;

        return "$f[5]-$f[4]-$f[3] $f[2]:$f[1]:$f[0]";
}
```

De esta forma cuando hagamos uso de nuestra función *sysdate*, por ejemplo:

```
print sysdate(), "\n";
```

esto nos devolverá la fecha con formato:

```
2009-02-05 22:00:00

        (En formato: Año-Mes-Día Hora:Minutos:Segundos)
```

Como el elemento del año nunca va a ser menor que 10, no debemos preocuparnos de él.

9.3.2 Ejercicio 2:
Monitorizar el tamaño de archivos log para que no crezca mas del tamaño permitido.

Planteamos la situación:

En los sistemas Operativos de Servidor, que son en su inmensa mayoría de tipo Unix o GNU/Linux, tenemos siempre archivos log que son archivos de registro de actividades, movimientos, alarmas, etc.

Como en la mayoría de las veces los mecanismos que escriben en esos archivos no se preocupan de administrar el tamaño de los mismos, debemos hacerlo nosotros para que en algún momento dado no tengamos unos archivos con el tamaño de varios GB que al momento de analizarlos es muy lento, así mismo su recorrido por los sistemas de administración de alarmas.

Generaremos entonces una subrutina que revise si tiene el tamaño máximo establecido (por nosotros mismos) y si es así en ese mismo momento le cambie el nombre a otro con terminación de número consecutivo y cree uno nuevo, escribiendo en ese último, que tiene tamaño cero.

Podríamos primero copiar el archivo a uno con terminación consecutiva y después vaciarlo, pero si mide muchos MB el proceso es lento y de nada nos sirve si luego lo vaciaremos.
Así que seleccionamos el primer método.

Para que esto funcione ágil deberemos contener el último consecutivo guardado en un archivo dedicado a eso, para no hacer una lista de los archivos y encontrar el último consecutivo.
Así cada vez que necesitemos saber cual es el último consecutivo solo leeremos ese archivos y sabremos cual es.

Comenzamos el ejemplo:

```
my $max_size = 10485760;      #10 MB         # el tamaño máximo otorgado
my $file_consecutivo = "./file.consec";      # el contenedor del consecutivo
my $file_log = "./myfile.log";               # el archivo log principal
my $mensaje = "Este es el mensaje a escribir en el archivo log";

if ( -s "$file_log" > $max_size ) {          # si pasa el umbral...
    open (FILE, "<$file_consecutivo");
      my ($consec) = <FILE>;                 # sabemos cual es el último
    close FILE;

    $consec += 1;                            # aumentamos el consecutivo en 1
    rename($file_log, "$file_log.$consec"); # movemos el archivo log
                                 # hacia otro con terminación consecutiva

    open (FILE, ">$file_consecutivo");
```

```
            print FILE "$consec";   # nuevo valor del último consecutivo
        close FILE;

        open(NEWLOG, ">$file_log");      # el archivo log vacío (nuevo)
          print NEWLOG $mensaje, "\n";   # imprimimos el mensaje nuevo
        close NEWLOG;
    }
```

De esta manera tenemos un archivo *.log y otro de tipo *.consec el cual contiene un número que representa el consecutivo de los archivos antiguos que son del tamaño de 10 MB, y que guardamos como respaldo.

Ejemplo:

alarmas.log
alarmas.log.1

Los mecanismos de análisis tendrán que leer tanto el actual *.log como el último respaldo para procesar la información debida.
Los demás podrán borrarse cada cierto tiempo.

Este material lo podremos colocar como una función propia al que le podamos "pasar" argumentos que son los nombres de los archivos, el tamaño máximo, etc.

```
    sub print_log {
            my $max_size = shift;
            my $file_consecutivo = shift;
            my $file_log = shift;
            my $mensaje = shift;

            if ( -s "$file_log" >= $max_size ) {

                open (FILE, "<$file_consecutivo");
                  my ($consec) = <FILE>;
                close FILE;

                $consec += 1;                    # aumentamos el consecutivo en 1
                rename($file_log, "$file_log.$consec"); # movemos el archivo log
                                                 # hacia otro con terminación consecutiva

                open (FILE, ">$file_consecutivo");
                  print FILE "$consec";   # nuevo valor del último consecutivo
                close FILE;

                open(NEWLOG, ">$file_log");      # el archivo log vacío (nuevo)
                  print NEWLOG $mensaje, "\n";   # imprimimos el mensaje nuevo
                close NEWLOG;
            }
    }
```

La subrutina la tendríamos que llamar de forma parecida a esto:

```
print_log($max_size, $file_consecutivo, $file_log, $mensaje);
```

Declarando antes las variables enviadas a la sub.

O talvez algo mas primitivo:

```
print_log(10485760, "./file.consec", "./myfile.log", "Mensaje X");
```

Claro, cada quien le pone los nombres de archivos y las terminaciones que necesita para trabajar.

Aclaremos que esto es un ejemplo y que se puede tomar como base, no es necesariamente que esto deba ser así para colocarlo en producción, talvez se necesite mover solamente a un archivo de respaldo y un original y no crear mas de una copia. Este mecanismo lo dejamos de tarea para el lector, además de los mecanismos de seguridad de archivos como los bloqueos. Porque los archivos log también deben protegerse con bloqueos.

9.3.3 Ejercicio 3:
Monitorizar el número de líneas de archivos log para que no crezca mas del número permitido.

Ahora, sin fijarnos en el tamaño en Bytes del archivo, veremos como podemos limitar el número de líneas de un archivo para ir borrando las primeras del mismo, porque ya no nos interesan debido a su antigüedad.
Como las últimas líneas son las mas nuevas, son las que nos interesan.

Husmeando un poco por el cpan (search.cpan.org), encontré un módulo existente en las distribuciones estandard de Perl que me permite trabajar con archivos como si fuera un array y cada línea, un elemento, su nombre: **Tie::File**.

Conviene al lector acercarse en este momento a la documentación de Tie::File para acostumbrarse poco a poco a leer esta documentación. Yo se muy bien que hasta este momento la mayoría de los lectores (y que empiezan a trabajar con Perl) no lo ha hecho.

Veamos como podemos trabajar con estos datos:
Primero declaramos nuestras variables:

```
my $max_lines = 5000; #             # máximo número de líneas
my $file_log = "./myfile.log";      # el archivo log principal
my $mensaje = "Este es el mensaje a escribir en el archivo log";
```

Ya sabemos que el nombre de una variable escalar solo puede contener letras, números o guión bajo, sin embargo existen variables especiales dentro de Perl como la del mensaje de error que ya vimos en el punto 8.5.3, **$!**.

También tenemos **$.**, la cual nos indica el número de línea en la que estamos posicionados, cuando un bucle recorre iterativamente un archivo abierto.

Así que esto nos va a servir para saber el número de líneas que tenemos en un archivo:

```
my $num_lines;        # nuestra variable global del número de líneas
open (FILE, "<$file_log");
while (<FILE>) {
    $num_lines = $.;  # cada vez que pasa una línea, cambia de valor...
                      # en la última vuelta queda grabada en la variable
}
close (FILE);
```

Ahora que tenemos el número de líneas, solo nos queda compararla con el tamaño máximo de líneas permitido y eliminar las sobrantes.

Es el momento de utilizar el módulo **Tile::File** del que hablamos, para hacer esto último.

```
my @contenido;        # declaramos la variable que ligaremos al archivo

my $oc = tie @contenido, 'Tie::File', "$file_log", memory => 0;

$oc->flock(LOCK_EX);  # bloqueamos el archivo con un método establecido
                      # LOCK_EX de exclusive, no lectura, no escritura

if (@contenido) {     # nos aseguramos que el archivo no esté vacío

    if ( $num_lines > $max_lines ) { # si pasa el umbral...

        my $dif = $num_lines - $max_lines;   # cuantas líneas de mas?

        for my $i ( 0 .. $dif ) {     # por cada línea de mas

            $contenido[$i] = "";      # eliminamos esas líneas
                                      # (vaciamos el elemento del array)
        }

    }
} else {
```

```
        print .= "Can't make TIE over \"$file_log\" \n";
    }

    $oc->flock(LOCK_UN);  # desbloqueamos el archivo

    untie @contenido;
```

La segunda línea no la cometamos porque vamos a analizarla un poquito mas a fondo.
Como comentamos en líneas anteriores, Tie::File "enlaza" un archivo físicamente a un array de nuestro script. Y cada elemento del array representa una línea del archivo.

Inicialmente creamos un objeto (**$oc**), resultado de hacer **Tie** sobre el archivo. Para posteriormente bloquearlo con los métodos que nos indican en la documentación de **Tie::File**.

Esa segunda línea tiene una sintaxis clásica según la documentación del módulo.
El último elemento de la lista de configuración de **Tie::File** es *memory => 0*.
Con esto hacemos que las modificaciones se realicen directamente en el archivo, y no en el buffer que maneja Tie.
O sea, que no asignamos memoria al buffer de **Tie::File**.
Ruego al lector nuevamente que lea bien la documentación de Tie::File en cpan.

De esta manera siempre tendremos un archivo con una cantidad de líneas requerido... ni una mas.

9.3.4 Ejercicio 4:
Monitorización de Filesystems.

Ya se que hicimos una pequeñas pruebas sobre este tema, pero podemos ampliarla un poco mas.
Lo que nos pedirán en nuestro trabajo probablemente será algo como esto:

Debemos tener monitorizado los File Systems de algunos servidores, de manera que, cada file system pueda tener un umbral diferente.
Además deberemos generar un log en algún archivo.
Finalmente deberemos poder enviar un email cuando la alerta suceda.

Aparentemente se ve complicado, pero no lo es, ya que hemos visto muchas cosas en el transcurso de nuestro pequeño libro.

Comencemos por establecer un archivo de configuración, al cual le podremos agregar mas líneas como variables deseemos tener, sin necesidad de abrir el archivo del script para modificarlo.

```
#!/usr/bin/perl
use strict;
$configfile = "./fs.conf";
```

Como queremos que el archivo tenga el formato convencional de la mayoría de los archivos de configuración:

```
# Comentario del archivo
VARIABLE = VALOR
```

Deberemos ser capaces de leer ese archivo y transformar sus líneas en variables legibles para nuestro script.

```
#!/usr/bin/perl
use strict;
use Fcntl ':flock';

my $configfile = "./fs.conf";

my %vars = ();                # declaramos nuestro hash para las variables
open(VARSENV, "<$configfile");
    while (1) { flock(VARSENV, LOCK_SH) or next; last; }   # bloqueo maza
                                                           # para lectura
while (<VARSENV>) {
    $_ =~ s/\s//g;            # quitamos los espacios
    $_ =~ s/\n//g;            # es lo mismo que chomp

    unless ( $_ eq "" || $_ =~ /^\#/ ) {           # si no es comentario
       my ($llave, $valor) = split(/\=/, $_);
           $vars{$llave} = $valor;
       }
}
    flock(VARSENV, LOCK_UN);
close VARSENV;
```

Dentro del script crearemos un hash para poder meter ahí las variables que queramos.
Lo bloqueamos.
Analizamos línea por línea. Le quitamos los espacios y los saltos de línea.
Si no contiene **#** al inicio, entonces separamos mediante el signo de **=**, lo que es la llave y el valor para luego crear el elemento del hash.

De esta forma, podremos tener en cualquier momento el valor de cada variable accesible para su proceso.

Por ejemplo, declaramos en el archivo de configuración la variable:

```
LOGFILE = almslog
```

Cuando queramos llamar a esta variable para saber su valor, solo tendremos que llamar la llave del hash %vars LOGFILE para saber su valor.

Como prueba y ejemplo imprimimos el valor de esa llave:

```
$vars{'LOGFILE'};     # imprime: alms.log
```

Nos imprime *'alms.log'*, pues es el valor de LA LLAVE *$vars{'LOGFILE'}*.

Veamos como podemos crearlo:

```
# Archivo de configuración de file systems
# Formato: Variable = Valor

# Archivo para guardar las alarmas
ALMLOG = /path/to/alms.log

# Archivo para guardar los errores de ejecución
ERRORLOG = /path/to/errores.log

# Crecimiento máximo de sistemas de archivos
# Para agregar mas sistemas de archivos a monitorizar
# solo agregue mas bloques en el formato:
# FILESYSTEMS = /path|umbral,/otro/path|umbral
# el umbral en %, ejemplo:
# FILESYSTEMS = /dev/sda1|80,/dev/sda3|50
FILESYSTEMS = /dev/sda1|80, /dev/sdb1|70

# recordemos que los espacios se eliminarán...

# Activar envío de email?
# 0 = no, 1 = sí
ENVIO_EMAIL = 0

# cuentas de email para notificar las alarmas
# separadas por comas (,)
LS_EMAIL = hugo.maza@gmail.com.mx
```

En nuestro caso, decidimos seccionar el valor de la variable FILESYSTEMS mediante el uso de comas, cada sistema de archivo a monitorizar está separado por comas '**,**', y cada sección de estas, también la separaremos por el pipe '|', así podremos saber qué porcentaje máximo debemos soportar antes de enviar una alarma.

Ya se imaginará el lector que usaremos la función **split** varias veces durante el script.
Y bueno, veamos como continuamos con el desarrollo.
Primeo vamos a crear un array con los sistemas de archivo extraídos en bruto, esto es, contienen tanto el file system como el umbral:

```
my @fs = split (/\,/, $vars{'FILESYSTEMS'});
```

Ahora crearemos un hash para tener cada filesystem como la llave y el umbral como el valor:

```
my %fshash;
foreach (@fs) {
    my($lla, $val) = split (/\|/, $_);      # separamos por pipe

    $fshash{$lla} = $val;                   # ej: (/home, 80)
}
```

Después (y esto ya lo vimos) ejecutaremos el comando *df -k* de Unix para conocer los sistemas de archivos y sus usos, y colocamos la salida en una variable, **$cmd**.
Esta misma salida la separamos en líneas y la colocamos en un array:

```
my $almtxt;
my $cmd = `df -k`;    # cargamos las líneas del comando a una variable

my @lns = split(/\n/, $cmd);         # separamos cada línea y las colocamos
                                     # como elementos de un array
```

Ahora comienza lo entretenido.

Por cada línea del comando, analizamos si tiene un file system asociado y si es así, usamos esta cadena para formar la llave del hash `%fshash`, lo cual nos indica que si existe esa llave, podremos proceder a comparar el valor de la carga de la salida del comando, con el valor del elemento del hash para saber si está dentro del umbral o no.

Veamos:

```
foreach my $linea ( @lns ) {         # por cada $linea del array @lns...

    $linea =~ /^([0-9a-zA-Z\/]+).*/; # obtenemos el primer campo
    my $sys = $1;                    # que es la parte del file system

    if ( $fshash{$sys} ) {           # si existe el sistema (la llave)...

        $linea =~ /.*\s([0-9]+)\%\s.*/;      # buscamos la columna y
        my $valor = $1;                      # extraemos el valor
```

```
            if ( $valor >= $fshash{$sys} ) {     # comparamos el valor con el
                                                 # umbral
                $almtxt .= "Alarma!: $sys en $valor\%. Igual o por
                                encima del umbral de $fshash{$sys}\% \n";
            }
        }
    }
```

Si queremos demostrar que funciona antes de proseguir a grabar las alarmas en un log y sobre todo, a enviar correos, en este punto del script podemos imprimir el resultado, configurando los umbrales mas abajo de lo que en la actualidad están.
Así sabremos que funciona:

```
print $almtxt;
```

Una vez que probamos que funciona (y debe funcionar), borramos esa línea de impresión y continuamos con el script.

```
if ( $almtxt ) {      # si hay alarmas
        my $fecha = sysdate();       # esta sub ya la establecimos antes

        # imprimimos en el archivo log:
        open(FILE, ">>$vars{"ALMLOG"}");
               # bloqueo maza para escritura:
               while (1) { flock(FILE, LOCK_EX) or next; last; }
        print FILE "$fecha--$almtxt";
        close FILE;

        # imprimimos en sendmail para enviar correo si está habilitado:
        # (si declaramos ENVIO_EMAIL = 0, no va a enviar correo)
        if ( $vars{"ENVIO_EMAIL"} ) {
            open(MAIL, "|/usr/sbin/sendmail -t");
            print MAIL "From: root\@miserver.com\n";
            print MAIL "To: $vars{'LS_EMAIL'}\n";
            print MAIL "Subject: Alarma de File System sobre el umbral\n\n";
            print MAIL "Con fecha $fecha, se creó alarma(s) de File System
                                                    sobre su umbral.\n";
            print MAIL "$almtxt\n\n";
            close MAIL;
        }
}

exit;
```

Y así, nuestro script envía alarmas vía sendmail además de registrar la misma en un archivo log.

9.3.4.1 Ejecución del script vía crontab

Ahora solo nos falta colocar el script en el crontab del sistema.
Cada 5 minutos o cada hora, dependiendo de la urgencia con que deba atenderse la alarma.

Si el servidor es un equipo con GNU/Linux, es muy probable que solo se necesite colocar la ruta del script en el crontab.

Pero si es algún sabor de Unix como Solaris, HP-UX, etc., como estos sistemas no cargan las variables de ambiente del usuario que ejecuta el crontab, es necesario cargar estas variables de ambiente antes de ejecutar el script.

Usualmente hacemos un ejecutable en shell que se ejecute en crontab, y este llama al script de Perl después de cargar las variables de ambiente, como el HOME y otros.

Ejemplo de entrada en el crontab para levantar el archivo en shell:

```
#### MONITOREO DEL FILESYSTEM . CADA HORA
1 * * * * /root/scripts/FS/fs.sh >/dev/null 2>&1
```

En el ejemplo anterior se ejecuta cada hora al minuto uno, todos los días.

Claramente hacemos recordar al lector, que este libro está dedicado a aquellas personas que administran sites y que tienen muy poca experiencia o nula en programación, pero que sí saben moverse en Servidores con Sistema Operativo POSIX compatible (Unix, Unix-Like).

El script bash, dependiendo de cual intérprete usemos, puede ser diferente... veamos.

En Shell Bash sobre Ubuntu Server o Debian con usuario root:

```
#!/bin/bash
. /root/.bashrc
export HOME=/root     # ESTO NO ES NECESARIO SI CORRE EN GNU/LINUX
/usr/bin/perl /root/scripts/FS/fs.pl
```

En Shell Bash sobre Red Hat o CentOS con usuario root:

```
#!/bin/bash
. /root/.bash_profile
#export HOME=/root    # NO LO USAMOS EN GNU/LINUX
/usr/bin/perl /root/scripts/FS/fs.pl
```

En Korn Shell sobre Solaris con usuario root:

```
#!/usr/bin/ksh
# Para Solaris
HOME=/root
. $HOME/.profile
/usr/local/bin/perl /root/scripts/FS/fs.pl
```

En C Shell sobre Solaris (Unix) con usuario root:

```
#!/usr/bin/csh
# Solaris
setenv HOME /root
source /root/.cshrc
/usr/local/bin/perl /root/scripts/FS/fs.pl
```

Este archivo ***fs.sh*** debe tener permisos claro, de ejecución para que pueda funcionar. Además, NO es recomendable que el script corra como usuario root.

\- - -

Otro aspecto que hay que tomar en cuenta es la cantidad de correos que recibiríamos si las alarmas comienzan el viernes por la tarde y nosotros no revisamos el correo hasta el lunes por la mañana.

Deberemos en ese caso poseer métodos de control de envío de alarmas.

Si una mencionada alarma ya se envió, no enviarlas mas, o bien, enviarla con tiempos mas largos que el normal del crontab.

Pero esta es otra historia.
El lector debe poner a funcionar sus neuronas y crear por ejemplo un archivo que lleve control de los sistemas de archivo alarmados y en que hora, y si ya no hay alarmas, eliminar esa entrada del archivo.

Es fácil, solo necesitamos un poco de práctica en desarrollo con Perl...

Una última aclaración sobre este ejercicio, esta es una solución que se me ocurrió mientras escribía el libro, anteriormente había escrito otra solución un poco diferente, y tal vez la próxima vez se me ocurra otra muy distinta.

Al trabajar con Perl, cada que hagamos algo, podremos hacerlo distinto a la primera vez, ya que en Perl, como su lema lo indica, hay mas de una manera de hacerlo.

9.3.5 Ejercicio 5:
Prácticas con el uso de módulos adicionales de Perl

Ahora que nos hemos adentrado un poco en el uso de módulos, vamos a ejercitarnos un poco en el uso de algunos, un poco mas complejos que los que ya hemos visto.

Seleccioné un módulo que no viene con la distribución estandard de Perl, precisamente para poder hacer que el lector practique un poco la instalación de estos.

Se trata de un paquete de Perl muy útil en un sin número de ocasiones, Net::SSH::Expect.

Un ejemplo de la utilidad de este módulo... la conectividad por SSH usando llaves públicas (esto es común entre los administradores de Unix) es fácil cuando se hace entre plataformas similares, pero si vamos a conectar por ejemplo, un equipo con GNU/Linux con otro con HP-UX usando SSH con llave automática de autenticación, es muy difícil echar a andar esta conexión.

Es en estos casos que nos sirve mucho este módulo.

Lo descargamos de search.cpan.org.

Al descomprimirlo y tratar de instalarlo usando las famosas instrucciones, perl Makefile.PL, make, make test y make install, nos debe arrojar un error en el que apreciaría mucho que tuvieras instalado el módulo Expect.

Pero este a su vez, depende de una o dos librerías mas, para quedar debidamente instalado.

Si estás trabajando sobre Ubuntu Server, esto de instalar módulos de Perl es casi como pedirle que lo instale por ti, pues todas las dependencias se resuelven "automágicamente".

Una vez que tengamos listo el módulo Net::SSH::Expect, vamos a limpiarnos las manos y el sudor de la frente, después de la "talacha" que realizamos en la instalación y procedemos a ejercitarnos en el uso de este módulo.

En la documentación del módulo Net::SSH::Expect habla de diferentes métodos de uso, uno de ellos es al mas básico, el de crear el nuevo objeto para trabajar sobre él, new, al que iremos agregándole propiedades y podremos usarlo en determinado momento, para correr comandos en forma remota.

```
my $scpe = Net::SCP::Expect->new;
```

En la misma documentación, vemos que podemos "pasarle" argumentos como métodos.

Estos, como veremos muchas veces en los módulos, pueden ser en forma de variable hash: (llave => valor, llave => valor):

```
my $host = '192.168.0.99';
my $user = 'usuario';
my $pass = 'pass101word';

# El método new:
my $ssh = Net::SSH::Expect->new(
                host            =>      $host,
                user            =>      $user,
                raw_pty         =>      1
        );
# Y conectamos:
$ssh->run_ssh() or die "No puedo iniciar el proceso SSH: $!\n";
```

Si eres usuario avanzado de Unix, habrás usado alguna vez expect, y como este, también el módulo de Perl Net::SSH::Expect espera de regreso en línea de comandos, algo con que trabajar:

```
$ssh->waitfor('password:\s*\z');     # El servidor pide el password
```

Una vez que el servidor ha respondido pidiendo el password, se lo damos:

```
$ssh->send($pass) or die "Password incorrecto\n";
```

Es en este momento cuando estamos ya conectados, y podremos enviar con este mismo método "send" mas comandos y esperar respuesta con el método "waitfor".

O bien, ejecutar en forma mas exacta los comandos de Unix y colocarlos en una variable para su análisis con el método “exec”.

Vamos a realizar un pequeño script que se conecte a un servidor lejano, se autentique con su password correcto y después verifique simplemente que algún archivo exista.
Podremos también hacer un **tail** (comando de Unix que lee las últimas líneas de un archivo) para obtenerlo en una variable y procesarla.

Hagamos la práctica completa:

```
#!/usr/bin/perl -w

use strict;
use Net::SSH::Expect;

my $host = '192.168.0.99';
my $user = 'usuario';
my $pass = 'password';
my $file = '/home/user/file.txt';

my $ssh = Net::SSH::Expect->new(
                host           =>      $host,
                user           =>      $user,
                raw_pty        =>      1
        );

$ssh->run_ssh() or die "No puedo iniciar el proceso SSH: $!\n";

$ssh->waitfor('password:\s*\z');
$ssh->send($pass) or die "Password incorrecto\n";

# Ejecutamos un comando simple de Unix para saber si existe un archivo:
my $stdout = $ssh->exec("ls $file");

# Si no regresa que no existe el archivo o directorio:
unless ( $stdout =~ /No such file or directory/ ) {
        # Imprimimos el final del archivo:
        my $tail = $ssh->exec("tail $file");
        print "$tail\n";
} else {
        print "El archivo no existe\n";
}

# Cerramos la conexión con el método "close"
$ssh->close();

exit;
```

Y con este pequeño script vemos que podemos conectarnos en forma desatendida de un usuario, revisar información y de paso, administrar en forma automática, un Servidor remoto.

9.3.6 Ejercicio 6:
Ordenamiento por campos de una matriz (Algoritmo de Maza)

Dadas las innumerables ocasiones en que tengamos la necesidad de ordenar los datos de un archivo de texto, haremos un ejercicio para ilustrar una forma ágil de realizar esta ordenación no solo por línea, sino por campo, si acaso hemos colocado la información del

archivo en una matriz de datos dentro de una variable en nuestro script de Perl.

Visualicemos una matriz de datos:

```
A0      A1      A2      A3      A4      A5      A6

B0      B1      B2      B3      B4      B5      B6

C0      C1      C2      C3      C4      C5      C6

D0      D1      D2      D3      D4      D5      D6
```

En esta matriz tenemos 4 líneas y dentro de cada una hay 7 campos, desde el campo 0 al 6.
Esto nos recuerda que podemos cargar esta matriz dentro de un array bidimensional, el cual estudiamos en el capítulo 3.5.

A raíz de que no encontré en la red grande (el maestro Internet) ningún método que me ayudara a ordenar una matriz de datos, o mejor dicho, un array bidimensional por medio del número del elemento que yo le pasara como argumento, decidí construir un método simple de *Ordenamiento de una matriz por medio del* ***algoritmo de Maza****:*

```
$campo = 2;                                     # número de campo a ordenar
                                                # desde 0 hasta n
@array = sort {

    ($a->[$campo] <=> $b->[$campo])             # comparamos números y los
                                                # ordenamos de mayor a menor.
       ||                                 # o si no son números...

    ($a->[$campo] cmp $b->[$campo])             # comparamos caracteres y los
                                                # ordenamos de mayor a menor
} @array;
```

Si bien cada uno de estos métodos son conocidos por todos, había que juntar todo para lograr realizar el propósito.

Dejo al lector un ejemplo de uso para que juegue con el y estudiar su comportamiento.

```
#!/usr/bin/perl
$arrayref = [
  ['Casa', 'Carro', 'Moto'],
  [24, 89, 500],
  ['Perro', 'Gato', 'Caballo'],
  ['Calle', 'Ciudad', 'Pais'],
  [22, 303, 405]
];
```

```
$campo = shift;

@{$arrayref} = sort {         # de-referenciación

    ($a->[$campo] <=> $b->[$campo])          # comparamos números y los
                                             # ordenamos de mayor a menor.
    ||                                       # o si no son números...

    ($a->[$campo] cmp $b->[$campo])          # comparamos caracteres y los
                                             # ordenamos de mayor a menor
} @{$arrayref};       # de-referenciamos

for $i ( 0 .. $#{$arrayref} ) {
        for $j ( 0 .. $#{$arrayref->[$i]} ) {
                print $arrayref->[$i][$j] , "\t";
        }
        print "\n";
}

exit;
```

Como podemos ver, el script debe recibir un argumento, que podamos pasar al script fácilmente en la línea de comandos ejecutando el script seguido de un espacio y el número de campo por el que deseamos ordenar la matriz de datos:

```
user@host:~$ ./ordenar.pl 0
```

La línea anterior ordenará la matriz según el campo 0, la primer columna

```
user@host:~$ ./ordenar.pl 2
```

Y la anterior línea ordenará según el campo2 , la tercer columna.

Ejemplos de los resultados:

```
user@host:~$ ./ordenar.pl 2
Perro   Gato    Caballo
Casa    Carro   Moto
Calle   Ciudad  Pais
22      303     405
24      89      500
user@host:~$ ./ordenar.pl 1
Casa    Carro   Moto
Calle   Ciudad  Pais
Perro   Gato    Caballo
24      89      500
22      303     405
```

Créditos:

Texto del libro “Manual de Perl básico desde cero”:
Hugo Maza Moreno

Dromedario de portada:
Carlos Angel Contreras Escajeda

Indautor. México. Mayo 2010.

Abril de 2009 – Septiembre de 2009
Revisión final para publicación: Febrero de 2010
Fecha de publicación de esta versión: 20 Abril de 2010