Ya hablamos en el capítulo anterior de la singularidad de Perl, en este capítulo vamos a hablar sobre la pluralidad. La pluralidad en Perl esta representada por las listas y los arreglos.

Una lista es una colección ordenada de escalares. Un array es una variable que contiene una lista. En Perl, estos dos términos se utilizan a menudo como si fueran intercambiables. Pero, para ser precisos, una lista es datos, y un array es una variable. En este sentido, puedes tener una lista de valores que no estén contenidos en un array, pero cada variable de tipo array contiene una lista (incluso si la lista esta vacía). Por ejemplo:

```
36
12.4
42
"hola"
1.72e30
"bye\n"
```

Cada elemento en un array o una lista, es una variable escalar independiente con un valor independiente. Estos valores están ordenados, es decir, tienen un secuencia particular desde el primer hasta el último elemento. Los elementos de un array o una lista están `indexados` por pequeños enteros comenzando por el cero En Perl, es posible cambiar el número de inicio de un array o una lista indexada. Larry luego considero esto como una mala característica y su (ab)uso esta fuertemente desaconsejado (0) y contando de uno en uno, entonces el primer elemento de cualquier array o lista es siempre el elemento cero (0).

Puesto que cada elemento es un valor escalar independiente, una lista o una matriz puede contener números, cadenas, valores undef, o cualquier mezcla de diferentes valores escalares. Sin embargo, es más común tener todos los elementos del mismo tipo, tales como una lista de títulos de libros (todas son cadenas) o una lista de cosenos (todos son números).

Las matrices o listas pueden tener cualquier número de elementos. La lista mas pequeña es aquella que no tiene elementos, mientras que la mas grande puede llenar toda la memoria disponible. Una vez mas esto es parte de la filosofía de Perl "No hay limites innecesarios".

Accediendo los Elementos de un Array

Si ha usado arrays en otro lenguaje, no se debe sorprender al encontrar que perl provee un mecanismo de subindice del array para referirse a un elemento por un indice numérico.

Los elementos de un array son enumerados usando una secuencia de enteros, comenzando en cero (0) e incrementando de uno en uno para cada elemento. Por ejemplo:

```
$prueba[0] = "yabba";
$prueba[1] = "dabba";
$prueba[2] = "doo";
```

El nombre del array en este caso "prueba", es de un espacio de nombres totalmente separado de los escalares, puedes tener una variable escalar llamada $prueba en el mismo programa, y perl va a tratarlas como cosas diferentes y no va a confundirse.

Puede usar un elemento de un array como por ejemplo $prueba[0] en cualquier lugar como si se tratara de un escalar como $prueba. Por ejemplo, puede obtener el valor de un elemento del array o cambiar su valor con algunas expresiones cortas y prácticas que vimos en el capítulo anterior:

```
print $prueba[0];
$prueba[2] = "diddley";
$prueba[1] .= "whatsis";
```

Por su puesto, el subindice puede ser cualquier expresión que devuelva un valor numérico. Si no es un entero, automáticamente va a ser truncado al próximo entero menor. Por ejemplo:

```
$number = 2.71828;
print $prueba[$number - 1]; # Es lo mismo que imprimir $prueba[1]
```

Si se indica un elemento en el subindice que este mas allá del final del array, el valor correspondiente va a ser `undef`. Al igual que con las variables escalares, si nunca ha guardado un valor en una variable escalar, pues entonces el valor de esta variable va a ser `undef`. Por ejemplo:

```
$blank = $prueba[ 142_857 ]; # obtengo undef
$blanc = $mel;               # escalar no usado $mel obtengo undef
```

Indices Especiales para un Array

Si guardas en un elemento del array que esta mas allá del final del array, este automáticamente se extiende hasta donde sea necesario, aquí no hay límite de longitud, el límite básicamente es la memoria que Perl tenga disponible para usar. Si Perl necesita crear elementos intermedios, estos son creados con valores undef. Por ejemplo:

```
$rocks[0] = 'bedrock';
$rocks[1] = 'slate';
$rocks[2] = 'lava';
$rocks[3] = 'crushed rock';
$rocks[99] = 'schist';      # ahora hay 95 elementos undef
```

Algunas veces necesitamos encontrar el ultimo elemento indexado en el array. Para el array `rocks` que hemos definido en el ejemplo anterior, el ultimo indice del array es `$#rocks`. Este valor no es igual a la cantidad de elementos contenidos en el array, esto es porque hay un indice 0.

```
$end = $#rocks;                     # 99
$numero_de_elementos = $end + 1;
$rocks[$#rocks] = 'hard rock';      # El último elemento.
```

El uso del valor $#name como un indice, como vimos en el ejemplo anterior, ocurre con tanta frecuencia que Larry proporciono un acceso directo: indices negativos para el array contando desde el final del array. No valla a pensar con esto que puede ir mas allá del inicio del array. Si tienes tres elementos en el array, entonces los indices negativos válidos son: -1 (el último elemento), -2 (el elemento del medio), y -3 (el primer elemento). En la práctica parece que no se suele usar esto, con la excepción del indice -1.

```
    $rocks[ -1 ] = 'hard rock'; # una forma fácil de hacer el ultimo
ejemplo
    $dead_rock = $rocks[-100];  # se obtiene 'bedrock'
    $rocks[ -200 ] = 'crystal'  # Error fatal !
```

Listas Literales

Un array (la forma de representar un valor de lista en el programa) es una lista de valores separados por coma encerrados entre paréntesis. Estos valores constituyen los elementos de la lista. Por ejemplo:

```
(1, 2, 3)   # Una lista de tres valores 1, 2 y 3
(1, 2, 3,)  # La misma lista de tres valores (la ultima coma se ignora)
("prueba", 4.5) # dos valores,  "prueba" y 4.5
( )         # una lista vacía, cero elementos
(1..100)    # Una lista de 100 enteros.
```

El ultimo ejemplo, aparece el operador de rango `..`. Este operador crea una lista de valores contando desde el escalar que se encuentra a la izquierda hasta el escalar de la derecha, uno a uno. Por ejemplo:

```
(1..5)      # lo mismo que (1, 2, 3, 4, 5)
```

```
(1.7..5.7)  # la misma cosa, pero se truncan los valores.
(5..1)      # Una lista vacía, solo se cuenta hacia arriba.
(0, 2..6, 10, 12) # Esto es (0, 2, 3, 4, 5, 6, 10, 12)
($m..$n)    # rango determinado por los valores de $m y $n
(0..$#rocks) # Usando el indice del array rocks.
```

Como puede ver en los últimos dos ejemplos, los elementos de una lista literal no necesariamente son constantes, pueden ser expresiones que van a ser evaluadas cada vez que el literal sea usado. Por ejemplo:

```
($m, 17)        # Dos valores, el valor correspondiente de $m y 7
($m+$om, $p+$q) # Dos valores
```

Por supuesto, la lista puede contener cualquier valor escalar, como esta lista de cadenas:

```
("walter", "lilibeth", "juan", "juan", "jose", "carlos")
```

El atajo qw

Resulta que las listas de palabras simples son necesarias con frecuencia en programas en Perl. El atajo qw hace que sea mas fácil generarlas sin tener que escribir un montón de comillas adicionales:

```
qw( walter lilibeth juan juan jose carlos );
```

qw significa "palabras citadas" del ingles "quoted words". Perl trata esto como una cadena entre comillas simples en el contexto escalar, de manera que no podrá usar `\n` o `$prueba` en una lista qw como si estuviera entre comillas dobles.

Los espacios en blanco (caracteres como espacios, tabs, y nuevas lineas) son descartados, y todo lo que queda se convierte en una lista de elementos. Tomando en cuenta que el espacio en blanco se descarta, podemos escribir la lista de la siguiente forma:

```
qw( walter
    lilibeth
    juan
    juan
    jose
    carlos
);
```

El ejemplo anterior usa paréntesis como delimitador, pero Perl actualmente permite que uses cualquier caracter de puntuación como delimitador. Algunos ejemplos mas comunes son:

```
qw( walter lilibeth juan juan jose carlos );
qw! walter lilibeth juan juan jose carlos !;
qw/ walter lilibeth juan juan jose carlos /;
qw# walter lilibeth juan juan jose carlos #;
qw{ walter lilibeth juan juan jose carlos };
qw[ walter lilibeth juan juan jose carlos ];
qw< walter lilibeth juan juan jose carlos >;
```

Si necesita incluir el delimitador usado como parte de una cadena en uno de los elementos, probablemente escogió el delimitador equivocado, pero incluso podría hacerlo, escapando el delimitador con un `\`. Ejemplo:

```
qw! yahoo\! google ask msn ! # include yahoo! as an element
```

Aunque el lema de perl es "Siempre Hay Mas De Una Forma De Hacer Las Cosas", se preguntara, para que necesito todas estas formas de citar. Bueno, mas adelante que hay otras formas de citar en Perl que son útiles en casos especiales. Por ejemplo, ahora mismo podría necesitar hacer una lista de nombres de archivos Unix, entonces podría hacerlo así:

```
qw{
    /usr/dict/words
    /home/elsanto/.vimrc
}
```

Asignación de listas

De la misma forma que asignamos escalares a variables, podemos asignar valores de listas a variables. Por ejemplo:

```
($walter, $lilibeth, $juan) = (25, 26, 30);
```

Las tres variables en la lista de la izquierda, reciben nuevos valores, como si los asignara individualmente. Dado que la lista se constituye antes del inicio de la asignación. Esto hace que sea fácil intercambiar los valores de dos variables en Perl:

```
($walter, $lilibeth) = ($lilibeth, $walter);
($betty[0], $betty[1]) = ($betty[1], $betty[0]);
```

Pero que ocurre si el numero de variables (a la izquierda del signo de =) no es igual al numero de valores (a la derecha del signo =) en una asignación de lista. Los valores extra son simplemente ignorados. De igual manera si ocurre lo contrario, las variables adicionales obtienen el valor `undef`.

```
($fred, $barney) = qw< flintstone rubble slate granite >;
($wilma, $dino) = qw[flintstone];       # $dino obtiene undef.
```

Ahora que puedes asignar listas, puedes crear un array de cadenas con una linea de código como esta:

```
($rocks[0], $rocks[1], $rocks[2]]) = qw/talc mica feldspar/;
```

Podemos referirnos a toda una matriz a través de una notación simple que posee Perl. Solo tiene que usar una arroba (@) antes del nombre del array (y sin corchetes de indices) para referirse a toda la matriz. Este sígil funciona a ambos lados del operador de asignación.

```
@rocks  = qw/ bedrock slate lava /;
@tiny   = ( );                      # una lista vacia
@giant  = 1..1e5;                   # una lista de 100.000 elementos
@stuff  = (@giant, undef, @giant);  # una lista de 200.001 elementos
```

`@tiny` es una lista conformada por cero elementos. (Esto en particular no pone undef dentro de la lista, pero lo podemos hacer de manera explicita como se puede ver con la lista `@stuff`).

El valor de una lista nueva a la que no se le ha asignado nada es `( )` de lista vacía. Así como los valores escalares nacen con el valor `undef` una lista nace con el valor `( )` (lista vacía).

Larry escogió el signo @ porque el lee $calar (scalar) y @rray (array), es una buena regla nemotécnica para recordar.

Los operadores push y pop

Usted puede agregar nuevos elementos al fina de un array simplemente guardando estos elementos

con un indice nuevo mas grande. Pero los verdaderos programadores de Perl no usan indices Esto por su puesto es una broma. Los indices en Perl no son una fortaleza del lenguaje. Si usas pop, push y operadores similares en lugar de usar indices, tu código va a ser generalmente más rápido. En las siguientes secciones explicaremos como trabajar con arrays sin usar indices.

Un uso común para un array es guardar información, en donde los nuevos valores se añaden y se eliminan por la parte derecha de la lista (Este es el final de la lista que contiene el ultimo elemento, con el indice mas alto). Estas operaciones ocurren con la suficiente frecuencia como para tener sus propios operadores especiales.

El operador pop saca el último elemento de la lista y lo retorna. Ejemplo:

```
    @array      = 5..9;
    $fred       = pop(@array);  # $fred = 9,  @array = (5, 6, 7, 8)
    $barney     = pop @array;   # $barney = 8,  @array = (5, 6, 7)
    pop @array;                 # @array = (5, 6) (El 7 fue descartado)
```

El último ejemplo usamos pop en un "contexto vacío", que es una manera elegante de decir que el valor de retorno no va a ninguna parte. No hay nada malo con el uso de pop de esta manera.

Si la matriz esta vacía, pop no hace nada y devuelve `undef`.

Seguro noto que pop puede usarse sin los paréntesis. Esta es una regla general en Perl, siempre y cuanto no cambie el sentido de la expresión, se pueden remover los paréntesis. Un estudiante avanzado, va a reconocer que esto es una tautología

El operador contrario es `push` que agrega elementos o una lista de elementos al final de un array. Por ejemplo:

```
    push(@array, 0);            # @array ahora tiene (5, 6, 0)
    push @array, 8;             # @array ahora tiene (5, 6, 0, 8)
    push @array,  1..10;        # @array ahora tiene 10 elementos mas.
    @others = qw/ 9 0 2 1 0/;
    push @array,  @others;      # @array ahora tiene 5 elementos nuevos
(19)
```

Note que el primer argumento de `push` o el único argumento que pop require debe ser una variable array. Hacer push y pop de listas literales no tiene ningún sentido.

Los operadores shift y unshift

Los operadores `push` y pop hacen cosas al final del array (o al lado derecho del array). De manera similar, `unshift` y `shift` realizan las operaciones correspondientes pero al principio del array (o al lado izquierdo del array). Aquí hay algunos ejemplos:

```
    @array = qw# walter lilibeth juan #;
    $m = shift(@array);         # $m = "dino", @array = ("fred", "barney")
    $n = shift @array;          # $n = "fred", @array = ("barney")
    shift @array;               # @array = (),  vacío
    $o = shift @array;          # $o = undef,  @array esta vacío
    unshift(@array, 5);         # @array ahora tiene un elemento en la
lista
    unshift @array, 4;          # @array = (4, 5)
    @others = 1..3;
    unshift @array, @others;    # @array = (1, 2, 3, 4, 5)
```

Interpolando Arrays en Cadenas

Así como los escalares, los valores array pueden ser interpolados en una cadena de dobles comillas. Los elementos de un array son automáticamente separados por espacios (Espacio en blanco es el separador por defecto contenido como valor en la variable especial $").

```
    @rocks = qw{ flintstone slate rubble };
    print "quartz @rocks limestone\n"; # imprime 5 rocas separadas por
blanco
```

No se agregan espacios en blanco antes o después de la interpolación, si usted quiere esto, tendrá que hacerlo a mano.

```
    print "Three rocks are: @rocks.\n";
    print "There's nothing in the parens (@empty) here.\n";
```

Si de casualidad olvida que los array se interpolan entre dobles comillas, se va a llevar una sorpresa cuando intente colocar un correo electrónico en una cadena.

```
    $email = "walter@covetel.com.ve;    # MAL! Intentara interpolar
@covetel
```

Para poder hacer esto, deberá usar comillas simples o escapar el caracter @.

```
    $email = "walter\@covetel.com.ve";  # Correcto
    $email = 'walter@covetel.com.ve'    # otra forma de hacerlo.
```

La estructura de control foreach

Regularmente es necesario procesar una lista completa, por lo que Perl proporciona una estructura de control que hace justamente esto. El ciclo `foreach` pasa por cada valor de la lista, realizando una iteración por cada valor. Por ejemplo:

```
    foreach $rock (qw/ bedrock slate lava /) {
        print "One rock is $rock.\n"; # Prints names of three rocks
    }
```

La variable de control (`$rock` en este ejemplo) toma un nuevo valor de la lista por cada iteración. En la primera corrida es "bedrock; y en la tercera corrida es "lava".

La variable de control no es una copia del elemento de la lista. Es el elemento en si. Lo que significa que si modificas la variable de control dentro del ciclo, vas a modificar el elemento en si. Por ejemplo:

```
    @rocks = qw/ bedrock slate lava /;
    foreach $rock (@rocks) {
        $rock = "\t$rock";
        $rock .= "\n";
    }
    print "The rocks are:\n", @rocks;
```

¿Cual es el valor de la variable de control cuando el ciclo ha terminado?. Es el mismo que tenia para cuando el ciclo comenzó. Perl automáticamente guarda y restaura el valor de la variable de control de un ciclo foreach. Esto significa que no hay que preocuparse por usar una variable que este en uso en otro lado del programa.

La variable mágica de Perl: $_

Si omite la variable de control del ciclo, Perl usa la variable mágica, $_. Esta es (usualmente) una variable escalar mas, solo que con un nombre no usual. Por ejemplo:

```
foreach (1..10){ # Usa por defecto $_
    print "La cuenta va por $_\n";
}
```

Aunque esta no es la única cosa mágica que hace Perl, es el hechizo mas común.

Existen otros casos en los que Perl va a usar la variable mágica $_, uno de los casos en los que esto ocurre es con `print`, va a imprimir $_ si no le damos otro argumento.

```
$_ = "Hola";
print;          #Imprime por defecto $_
```

El operador reverse

El operador `reverse` toma una lista de valores (que pueden venir de un array) y devuelve la lista en el orden opuesto. Entonces si no le gusta esto de que el operador de rango `..` solo cuenta hacia arriba, esta es una forma de arreglar esto:

```
@lista = 6..10;
@lista2 = reverse(@lista);
@lista3 = reverse 6..10;
@lista = reverse @lista;
```

La última linea del ejemplo es importante, porque usa `@lista` dos veces. Esto es debido a que los operadores de lista tienen una fuerte precedencia a la derecha y una poca precedencia a la izquierda, y en la tabla de precedencias, los operadores de lista están de primeros junto a los paréntesis.

Recuerde que `reverse` devuelve la lista en el orden inverso, no afecta a la lista argumento, si el valor devuelto no se asigna a nada, es una operación inútil.

```
reverse @lista;             # MAL! no hace nada.
@lista = reverse @lista;    # Bien!
```

El operador sort

El operador sort toma una lista de valores (que pueden ser un array) y los ordena usando la ordenación interna de caracteres. Para cadenas ASCII, va a usar un orden ASCIIbetico. Como ya debe saber, ASCII es un lugar extraño donde todas las letras mayúsculas van delante de todas las letras minúsculas, donde los números vienen antes de las letras, y los signos de puntuación, esos signos que están aquí allá y en todas partes. Pero el orden ASCII es el orden por defecto, en otro capítulo vamos a ver como ordenar de otras maneras.

```
    @rocks      = qw / bedrock slate rubble granite /;
    @sorted     = sort(@rocks);             # bedrock, granite, rubble,
slate
    @back       = reverse sort @rocks;      # va de slate a bedrock
    @rocks      = sort @rocks;              # ordena y asigna
    @numbers    = sort 97..102;             # 100, 101, 102, 97, 98, 99
```

Contexto Escalar y Contexto de Lista

Esta es la sección mas importante de este capítulo. En efecto, esta es la sección mas importante del curso. No es una exageración afirmar que el desempeño de su carrera en el uso de Perl va a depender de comprender de manera correcta esta sección.

No significa que esto es difícil de entender. Es una idea simple: Una expresión dada puede significar cosas distintas dependiendo de en que lugar aparece. Esto no debería ser nada nuevo para usted, pasa todo el tiempo en lenguajes naturales. Por ejemplo en el contexto gocho hay expresiones que son contrarias al contexto caraqueño.

El contexto se refiere al lugar en donde se encuentra la expresión. Es la manera en que Perl interpreta las expresiones. Siempre se espera un valor de lista o un valor escalar. Ejemplo:

```
42 + algo # algo debe ser un escalar
sort algo # algo debe ser una lista.
```

Incluso si algo es exactamente la misma secuencia de caracteres, en un caso puede ser un simple valor escalar, mientras que en otro caso, puede ser una lista. Las expresiones en Perl siempre devuelven el valor apropiado para el contexto. Por ejemplo, tome un array, en un contexto de lista, este devuelve una lista de elementos. Pero en un contexto escalar, devuelve el numero de elementos que están contenidos en el array. Por ejemplo:

```
@people = qw ( walter lilibeth juan );
@sorted = sort @people;     # Contexto de lista: walter, lilibeth, juan
$number = 42 + @people;     # Contexto escalar: 42 + 3
```

Incluso en una asignación simple, ocurren diferentes contextos:

```
@list = @people;    # una lista de 3 personas
$n = @people;       # el número 3.
```

Usando Expresiones de listas en contexto escalar

Hay varias expresiones que deberían usarse para producir una lista. Si usa una de estas expresiones en un contexto escalar, ¿Qué cree que obtendrá?. Para saber esto debe ver que es lo que piensa el autor de la expresión. Usualmente esta persona es Larry, y usualmente en la documentación obtendrá la información completa. En efecto, parte del aprendizaje de Perl es aprender como es que Larry piensa.

Hay algunas expresiones que en contexto escalar no devuelven nada. Por ejemplo, `sort` en contexto escalar devuelve `undef`.

Otro ejemplo es `reverse`. En contexto de lista, devuelve la lista inversa. En contexto escalar, devuelve la cadena reversa (o el resultado reverso concatenando todas las cadenas). Ejemplo:

```
@backwards = reverse qw/ yabba dabba doo /; #  doo, dabba, yabba
$backwards = reverse qw/ yabba dabba doo /; #  oodabbadabbay
```

Algunos contextos mas comúnes son:

```
$prueba = algo  # contexto escalar
@gente = algo   # contexto de lista
($nombre,  $apellido) = algo # contexto de lista
($algo) = algo      # contexto de lista aún.
```

Aquí tenemos algunas otras expresiones que resultan en contexto escalar:

```
$fred = something;
$fred[3] = something;
123 + something something + 654
if (something) { ... }
while (something) { ... }
$fred[something] = something;
```

Y aquí tenemos algunas expresiones que proveen un contexto de lista:

```
@fred = something;
($fred, $barney) = something;
($fred) = something;
push @fred, something;
foreach $fred (something) { ... }
sort something
reverse something
print something
```

Usando Expresiones escalares en contexto de lista

En este sentido es mas claro, si una expresión no tiene normalmente un valor de lista, el valor escalar es automáticamente promovido a ser un elemento de la lista. Por ejemplo:

```
@fred = 6 * 7;  # Un elemento (42)
@barney = "hello" . ' ' . "world";
```

Hay algunos inconvenientes posibles:

```
@wilma = undef; # OOPS!
@betty = ( );   # la forma correcta de declarar una lista vacía.
```

`undef` es un valor escalar, asignar undef a un array no limpia el array. La manera correcta de limpiar un array es asignándole una lista vacía.

Forzar el contexto escalar

En ocasiones es necesario forzar el contexto escalar, en donde Perl espera una lista. En este caso, se puede usar la función falsa `scalar`. Esta no es una función verdadera, ya que lo único que hace es decirle a Perl que provea un contexto escalar.

```
@rocks = qw( talc quartz jade obsidian );
print "How many rocks do you have?\n";
print "I have ", @rocks, " rocks!\n";           #MAL!
print "I have ", scalar @rocks, " rocks!\n";    #BIEN!
```

De manera contraria, no hay una función que force el contexto de lista, confié en nosotros, no la va a necesitar.

STDIN en contexto de lista

En un capítulo anterior vimos que STDIN en un contexto escalar devuelve la ultima linea leida de la entrada estándar. En contexto de lista, este operador devuelve todas las lineas desde el principio hasta el final del archivo. Cada linea es retornada como un elemento separado de la lista. Por ejemplo:

```
@lines = <STDIN>; # lee la entrada estandar en el contexto de lista
```

Cuando la entrada estandar es un archivo, esto va a leer el archivo completo hasta el final. Pero cuando se lee de la entrada estándar en Unix, debe usar Ctrol + D.

Ejercicios

1. Escriba un programa que lea una lista de caracteres separados por lineas e imprima esta lista en modo inverso, si la entrada proviene del teclado, se necesita una señal para la finalización de la entrada, en Unix es CTRL-D.

2. Escriba un programa que lea una lista de números separados por lineas y por cada número imprima el nombre correspondiente a la lista mostrada a continuación. fred betty barney dino wilma pebbles bamm-bamm. Por ejemplo, si la entrada es 1, 2, 4, la salida debe mostrar lo siguiente, fred, betty, dino.

3. Escriba un programa que lea una lista de caracteres separados por lineas e imprima esta lista en orden ASCIIbetico, por ejemplo si la entrada fue fred, barney, wilma, betty, la salida debe ser, barney, betty, fred, wilma