Curso Perl Nivel Básico

Cooperativa Venezolana de Tecnologías Libres R.S.

31 de enero de 2011

info@covetel.com.ve

Índice general

1. Introducción **5**
- 1.1. Sobre Perl . . . 5
- 1.2. ¿ Porque Larry creo Perl ? . . . 5
- 1.3. ¿ Porque Larry simplemente no uso otro lenguaje ? . . . 5
  - 1.3.1. Características principales de Perl . . . 6
- 1.4. ¿ Perl es fácil o difícil ? . . . 6
- 1.5. ¿ Como se hizo Perl tan popular ? . . . 7
- 1.6. ¿ Que esta pasando con Perl ahora ? . . . 7
- 1.7. ¿ Que es CPAN ? . . . 8
- 1.8. ¿ Como hacer un programa en Perl ? . . . 8
- 1.9. Un programa simple . . . 8
- 1.10. Sobre los ejercicios y sus respuestas . . . 8

2. Datos Escalares **9**
- 2.1. Números . . . 9
  - 2.1.1. Todos los números internamente tienen el mismo formato . . . 9
  - 2.1.2. Literales de punto flotante . . . 9

3. Listas y Arreglos **11**
- 3.1. Accediendo los Elementos de un Array . . . 11
- 3.2. Indices Especiales para un Array . . . 12
- 3.3. Listas Literales . . . 13
- 3.4. El atajo qw . . . 13
- 3.5. Asignación de listas . . . 14
- 3.6. Los operadores push y pop . . . 15
- 3.7. Los operadores shift y unshift . . . 16
- 3.8. Interpolando Arrays en Cadenas . . . 16
- 3.9. La estructura de control foreach . . . 17
- 3.10. La variable mágica de Perl: $_ . . . 17
- 3.11. El operador reverse . . . 18
- 3.12. El operador sort . . . 18
- 3.13. Contexto Escalar y Contexto de Lista . . . 18
  - 3.13.1. Usando Expresiones de listas en contexto escalar . . . 19
  - 3.13.2. Usando Expresiones escalares en contexto de lista . . . 20
- 3.14. Forzar el contexto escalar . . . 20
- 3.15. STDIN en contexto de lista . . . 20

3.16. Ejercicios 21

4. Subrutinas **23**
4.1. Definiendo una Subrutina 23
4.2. Llamar una Subrutina 24
4.3. Retornar Valores 24
4.4. Argumentos 25
4.5. Variables privadas en Subrutinas 26
4.6. Longitud de la lista de parámetros 27
4.6.1. Subrutina &max bien escrita 27
4.7. Notas sobre Variables Léxicas (my) 28
4.8. Usando el Pragma Strict 28
4.9. El operador return 29
4.10. Omitir el signo & 30
4.11. Retornando valores no escalares 30
4.12. Variables Privadas Persistentes 31
4.13. Ejercicios 32

5. Entrada y Salida **35**
5.1. Entrada desde la Entrada Estándar 35
5.2. Invocación de Argumentos 37
5.3. Salida hacia la Salida Estándar 38
5.4. Formatear la salida con printf 39
5.5. Arrays y printf 40
5.6. Filehandles 40
5.6.1. Abrir un Filehandle 41
5.6.2. Filehandles Malos 42
5.6.3. Cerrar un Filehandle 42
5.7. Errores fatales con die 42
5.8. Mensajes de Advertencia con warn 43
5.9. Usando los Filehandle 43
5.9.1. Cambiar un Filehandle de salida por defecto 44
5.9.2. Reabrir un Filehandle por defecto 44
5.10. Salida con say 45

1 Introducción

1.1. Sobre Perl

Perl tiene unos 23 años hasta ahora. El lenguaje ha evolucionado desde una herramienta simple para la administración de sistemas tomando cosas de shell scripting y C (Perl 1) hasta convertirse en un poderoso lenguaje de propósito general (Perl 5), consistente, coherente, cambiando el paradigma de programación en general con la intensión de mantenerse por otros 25 años. (Perl 6)

Aun así, la mayoría de los programas en el mundo, escritos en Perl 5, aprovechan muy poco las características del lenguaje. Usted `puede` escribir programas en Perl 5 como si estuviera en Perl 4 (o Perl 3 o 2 o 1), pero los programas que aprovechan todo el increíble poder del mundo de Perl 5 que la comunidad ha inventado, y mejorado, son mas cortos, rápidos, mas poderosos, y fáciles de mantener que sus alternativas en anteriores versiones.

1.2. ¿ Porque Larry creo Perl ?

Larry crea Perl a mediados de 1980, cuando intentaba producir reportes sobre la información ordenada jerárquicamente en archivos de la red de noticias USENET para un sistema de reporte de bugs, y awk no dio la talla. Entonces Larry, como buen programador perezoso que es, decide hacer una herramienta que pueda usar para resolver este problema y que pueda volver a usar para otro problema similar en otro lugar, en otro tiempo.

1.3. ¿ Porque Larry simplemente no uso otro lenguaje ?

En este momento hay gran variedad de lenguajes, pero en esa época Larry no encontró un lenguaje que se adaptara a sus necesidades.

Si uno de los lenguajes de hoy, hubiese existido para la época, pues tal vez Larry lo hubiera usado para resolver el problema.

Necesitaba algo con la rapidez de codificación disponible en **Shell** o **AWK** y con algo del poder de las herramientas como **grep**, **cut**, **sort**, y **sed**, sin tener que recurrir a un lenguaje como **C**.

Perl, intenta llenar la brecha entre los lenguajes de bajo nivel como C, C++, y Assembler, y los lenguajes de alto nivel de la época como Shell Scripting. La programación a bajo nivel, usualmente es difícil y fea de escribir, pero rápida y sin limitaciones, es difícil superar la velocidad de un programa bien escrito a bajo nivel en una máquina dada. Y no hay mucho que se pueda hacer.

En el otro extremo, la programación convencional de alto nivel, tiende a ser lenta, dura, fea y limitada. Hay varias cosas que no puedes hacer con programación en shell o batch.

Perl es fácil, casi ilimitado, muy rápido y solo un poco feo.

1.3.1. Características principales de Perl

Perl es fácil

Cuando nos referimos que Perl es fácil, hablamos de que es fácil de usar, no es especialmente fácil de aprender. Si usted conduce un carro, que paso semanas o meses aprendiendo a conducir, entonces luego va a ser fácil de conducir. De igual manera cuando pase la misma o mas cantidad de horas aprendiendo Perl, entonces despues va a ser muy fácil de usar.

Perl es casí ilimitado

Hay muy pocas cosas que no podrás hacer con Perl. Seguramente usted no deseará escribir un controlador de algún dispositivo de interrupción a nivel de micro-kernel (A pesar de que ya alguien hizo esto). Pero la mayoria de las cosas que la gente común y no tan común necesita, la mayoria del tiempo son buenas tareas para Perl, desde scripts muy pequeños, hasta aplicaciones de nivel industrial.

Perl es sobre todo rápido

Esto es porque no hay nadie en el desarrollo que no lo utilice, de modo que entonces todos queremos que Perl sea rápido. Si alguien quiere añadir una nueva característica que sea genial, pero que introduzca lentitud a otros aspectos, es casí seguro que Larry va a rechazar la nueva característica hasta que no se encuentre una manera de escribirla de manera eficiente.

Perl es un poco feo

Esto es cierto, el camello se ha convertido en el simbolo de Perl, desde la cubierta del legendario libro **The Camel Book**, conocido tambien como **Programming Perl by Larry Wall, Tom Christianesen, and Jon Orwant**. Los camellos tambien son un poco feos, pero trabajan duro, incluso en condiciones difíciles. Los camellos siempre están ahí siempre para hacer el trabajo a pesar de todas las dificultades, incluso cuando se ven y huelen mal e incluse le escupen. Perl es un poco así.

1.4. ¿ Perl es fácil o difícil ?

Perl es fácil de usar, pero a veces difícil de aprender. Esto por su puesto es una gereralización. En el diseño de Perl, Larry tuvo que hacer muchas consideraciones. Cuando ha tenido la oportunidad de hacer algo más fácil para el programador a costa de ser más

difícil para el estudiante, ha decidido casi todo el tiempo a favor del programador. Esto es así, porque usted va a aprender Perl una sola vez, pero lo vas a usar una y otra vez. Perl posee cualquier cantidad de comodidades que le permiten al programador ahorrar tiempo. Por ejemplo, la mayoria de las funciones tienen un valor predeterminado, con frecuencia, el valor por defecto es justamente la manera en la que usted desea usar la función. De esta manera, podrás encontra lineas de código en Perl como las siguientes:

```
1    while (<>) {
2        chomp;
3        print join("\t", (split /:/)[0, 2, 1, 5] ), "\n";
4    }
```

En su totalidad, sin usar los atajos y los valores por defecto de Perl, este trozo de código seria aproximadamente 10 o 12 veces mas largo, entonces tomaria mas tiempo para escribirlo y leerlo. Va a ser mas duro de mantener y depurar cuando tenga mas variables.

1.5. ¿ Como se hizo Perl tan popular ?

Despues de jugar un rato con Perl, agregando cosas aquí y allá, Larry se lanzó a la comunidad de lectores de la Usenet, comunmente conocidad como **La Red**. Los usuarios de esta flota fugitiva que trabajaban en sistemas en todo el mundo (decenas de miles de ellos), le dieron retroalimentación, pidiendo a Larry, maneras de hacer esto, aquello o lo otro, muchos de los cuales Larry nunca había imaginado en su hasta el momento poco manejo de Perl.

Pero como resultado, Perl creció, y creció y creció. Creció en características. Creció en portabilidad. Lo que antes era un lenguaje poco disponible en solo un par de sistema tipo Unix, ahora tiene miles de páginas de documentación en linea gratis, docenas de libros, y miles de listas de correo en cientos de idiomas con un número incontable de lectores, e implementaciones en varios sistemas de uso cotidiano hoy en día, y adicionalmente un curso como este.

1.6. ¿ Que esta pasando con Perl ahora ?

Durante estos días, Larry Wall no escribe código, pero el sigue guiando el desarrollo y toma las desiciones importantes. Perl es mayormente mantenido por un grupo de resistentes y valientes personas llamadas **The Perl 5 Porters**. Puede suscribirse a la lista de correo en **perl5-porters@perl.org** para seguir el trabajo de esta gente y sus debates. Al momento de escribir esta documentación, muchas cosas estan pasando con Perl. Durante los últimos años, muchas personas han estado trabajando en la próxima versión de Perl: Perl 6.

No tire a un lado Perl 5, este sigue siendo la versión actual y estable. No se espera una version estable de Perl 6 durante un rato largo. Perl 5 hace todo lo de siempre, y siempre sera así. Perl 5 no va a desaparecer cuando Perl 6 salga. Y la gente puede seguir usando Perl 5 durante muchos años mas.

1.7. ¿ Que es CPAN ?

CPAN es **Comprehensive Perl Archive Network**, su primera parada para Perl. Posee el código fuente de Perl, listo para instalar y los **ports** para todos los sistemas no Unix, ejemplos, documentación, y extensiones del lenguaje.

CPAN es replicado en cientos de máquinas al rededor del mundo, comenzando en **http://search.cpan.org** o **http://kobesearch.cpan.org** para navegar o buscar un paquete. Si no tienes acceso a la red, todo CPAN puede caber en 8 o 10 GB. Es importante mencionar que debido a los cambios diarios en CPAN, un mirror con dos años sin actualizares es definitivamente una antiguedad.

1.8. ¿ Como hacer un programa en Perl ?

Ya es tiempo de hacer esta pregunta (aunque usted no la halla hecho aun), Los programas de Perl son archivos de texto plano, usted puede crearlos y editarlos en su editor de textos preferido, yo personalmente prefiero **VIM**. (Usted no necesita entornos de desarrollo especiales, aunque hay disponibles editores comerciales de varios proveedores. Nunca hemos utilizado uno de estos lo suficiente como para recomendarlo). Por lo general debería usar un editor de texto pensado para programadores, en lugar de un editor de textos convencional. ¿ Cual es la diferencia ? Bueno, un editor pensado para programadores le va a permitir hacer cosas que necesitan los programadores, como sangrado automático o subrayado de bloques de código, o indicar automáticamente la llave de cierre de un bloque de código. En sistemas operativos tipo Unix, los editores de texto mas populares son **Emacs** y **Vim**. BBEdit y TextMate son buenos para Mac OS X, mucha gente ha hablado muy bien sobre UltraEdit y PFE en Windows. La página del manual perlfaq2 tiene una lista de otros editores que también pueden usarse.

1.9. Un programa simple

Según la regla mas antigua de los libros de computación, cualquier libro sobre algún lenguaje de programación que tenga raices Unix, va a comenzar con **Hola, Mundo**, por ejempo en Perl seria así:

```
1  #!/usr/bin/perl
2  print "Hello, world!\n";
```

1.10. Sobre los ejercicios y sus respuestas

Los ejercicios se encuentran al final de cada capítulo, una vez que se desarrolle cada capítulo, se tomaran 45 a 50 minutos para resolver los ejercicios correspondientes a cada capítulo.

2 Datos Escalares

2.1. Números

Aunque los números y las cadenas con frecuencia se pueden tratar como escalares, es útil observarlos inicialmente por separado. Vamos a ver primero los números y luego pasamos a las cadenas.

2.1.1. Todos los números internamente tienen el mismo formato

Como vera en los próximos párrafos, se puede especificar tanto números enteros (como 255 o 2001) y números de coma flotante (números reales con puntos decimales, como 3.1416 o 1.35 x 1025). Pero internamente, Perl calcula valores punto flotantes de doble precisión. Esto significa que no hay valores enteros internamente en Perl. Una constante entera en Perl es tratada como su valor equivalente en coma flotante. Probablemente no se dará cuenta de la conversión, deje de buscar las distintas operaciones con enteros (en oposición a las operaciones de punto flotante), porque simplemente no existen. [1]

2.1.2. Literales de punto flotante

Un literal es la forma en que un valor se representa en el código fuente en Perl. Un literal no es el resultado de una operación de calculo o de una operación de I/O. Es datos escritos directamente en el código fuente.

Un literal de coma flotante, ya debe serle familiar. Números con y sin punto flotante también son permitidos (incluyendo el prefijo opcional mas y menos), y de remate el indicador de notación exponencial E.

```
1.25
255.000
255.0
7.25e45

```

[1] Existe un pragma llamado Integer, que permite realizar operaciones con enteros en lugar de punto flotante, pero es otra cosa, y no es de lo que estamos hablando en este punto

3 Listas y Arreglos

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:

```
1    36
2    12.4
3    42
4    "hola"
5    1.72e30
6    "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 [1] (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".

3.1. 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:

[1] 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

```
1     $prueba[0] = "yabba";
2     $prueba[1] = "dabba";
3     $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:

```
1     print $prueba[0];
2     $prueba[2] = "diddley";
3     $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:

```
1     $number = 2.71828;
2     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:

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

3.2. 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:

```
1     $rocks[0] = 'bedrock';
2     $rocks[1] = 'slate';
3     $rocks[2] = 'lava';
4     $rocks[3] = 'crushed rock';
5     $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.

```
1     $end = $#rocks;                    # 99
2     $numero_de_elementos = $end + 1;
3     $rocks[$#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.

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

3.3. 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    (1, 2, 3)   # Una lista de tres valores 1, 2 y 3
2    (1, 2, 3,)  # La misma lista de tres valores (la ultima coma se ignora)
3    ("prueba", 4.5) # dos valores,  "prueba" y 4.5
4    ( )         # una lista vacía, cero elementos
5    (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    (1..5)      # lo mismo que (1, 2, 3, 4, 5)
2    (1.7..5.7)  # la misma cosa, pero se truncan los valores.
3    (5..1)      # Una lista vacía, solo se cuenta hacia arriba.
4    (0, 2..6, 10, 12) # Esto es (0, 2, 3, 4, 5, 6, 10, 12)
5    ($m..$n)    # rango determinado por los valores de $m y $n
6    (0..$#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:

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

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

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

3.4. 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:

```
1    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:

```
1    qw( walter
2        lilibeth
3        juan
4        juan
5        jose
6        carlos
7    );
```

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

```
1    qw( walter lilibeth juan juan jose carlos );
2    qw! walter lilibeth juan juan jose carlos !;
3    qw/ walter lilibeth juan juan jose carlos /;
4    qw# walter lilibeth juan juan jose carlos #;
5    qw{ walter lilibeth juan juan jose carlos };
6    qw[ walter lilibeth juan juan jose carlos ];
7    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:

```
1    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:

```
1    qw{
2        /usr/dict/words
3        /home/elsanto/.vimrc
4    }
```

3.5. Asignación de listas

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

```
1    ($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.

3.6. 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 [2]. 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)
```

[2] 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

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. [3]

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

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

3.7. 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:

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

3.8. 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 `$''`).

```
1    @rocks = qw{ flintstone slate rubble };
2    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.

[3] Un estudiante avanzado, va a reconocer que esto es una tautología

```
1     print "Three rocks are: @rocks.\n";
2     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.

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

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

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

3.9. 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:

```
1     foreach $rock (qw/ bedrock slate lava /) {
2         print "One rock is $rock.\n"; # Prints names of three rocks
3     }
```

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:

```
1     @rocks = qw/ bedrock slate lava /;
2     foreach $rock (@rocks) {
3         $rock = "\t$rock";
4         $rock .= "\n";
5     }
6     print "The rocks are:\n", @rocks;
7
```

¿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.

3.10. 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:

```
1     foreach (1..10){ # Usa por defecto $_
2         print "La cuenta va por $_\n";
3     }
```

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.

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

3.11. 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:

```
1     @lista = 6..10;
2     @lista2 = reverse(@lista);
3     @lista3 = reverse 6..10;
4     @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.

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

3.12. 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.

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

3.13. 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:

```
1     42 + algo # algo debe ser un escalar
2     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:

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

Incluso en una asignación simple, ocurren diferentes contextos:

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

3.13.1. 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:

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

Algunos contextos mas comúnes son:

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

Aquí tenemos algunas otras expresiones que resultan en contexto escalar:

```
1         $fred = something;
2         $fred[3] = something;
3         123 + something something + 654
4         if (something) { ... }
5         while (something) { ... }
6         $fred[something] = something;
```

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

```
1    @fred = something;
2    ($fred, $barney) = something;
3    ($fred) = something;
4    push @fred, something;
5    foreach $fred (something) { ... }
6    sort something
7    reverse something
8    print something
```

3.13.2. 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:

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

Hay algunos inconvenientes posibles:

```
1    @wilma = undef; # OOPS!
2    @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.

3.14. 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.

```
1    @rocks = qw( talc quartz jade obsidian );
2    print "How many rocks do you have?\n";
3    print "I have ", @rocks, " rocks!\n";           #MAL!
4    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.

3.15. 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:

```
1    @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.

3.16. 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 seï¿½al para la finalizaciï¿½n de la entrada, en Unix es CTRL-D.
2. Escriba un programa que lea una lista de nï¿½meros separados por lineas y por cada nï¿½mero imprima el nombre correspondiente a la lista mostrada a continuaciï¿½n. 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

4 Subrutinas

Hasta el momento hemos visto y usado algunas funciones internas de Perl, como `chomp`, `reverse`, `print`, y otras mas. Al igual que otros lenguajes, Perl tiene la habilidad de crear `subrutinas`, que son funciones definidas por el usuario. [1] Pero una subrutina es siempre definida por un usuario, mientras que una función puede o no serlo. Esto es, la palabra *función* puede ser usada como sinónimo para *subrutina*, pero es una función que el programador puede definir, no las funciones internas de Perl.

El nombre de una subrutina es otro identificador de Perl (letras, dígitos y pisos bajos, pero este no puede iniciar con un dígito.) con un signo & (ampersand) a veces opcional adelante. Existe una regla acerca de cuando podemos omitir el signo & y cuando no, al final del capítulo vamos a ver esta regla. Por ahora vamos a usar el signo & adelante siempre, esto no puede olvidarse.

El nombre de una subrutina viene de un espacio de nombres separado, por lo que Perl no va a confundirse si usted tiene una subrutina llamada &fred y un la variable escalar llamada $fred en el mismo programa, aunque no hay una razón para hacer esto en circunstancias normales.

4.1. Definiendo una Subrutina

Para definir su propia subrutina, use la palabra reservada `sub`, el nombre de la subrutina (sin el signo &), luego el bloque de código identado (entre llaves)[2], va a conformar el cuerpo de la subrutina, quedaría algo como esto:

```
1    sub marino {
2        $n += 1;    # Variable global $n
3        print "Hola, marinera número $n!\n";
4    }
```

Las subrutinas se pueden definir en cualquier lugar del programa, pero los programadores que vienen de lenguajes como C o Pascal, les gusta colocar las subrutinas al inicio del archivo. Otros prefieren poner las subrutinas al final del archivo, entonces la parte principal del programa queda al principio. Esto depende de usted. En cualquier caso, normalmente no va a necesitar ningún tipo de declaración previa [3]

[1] En Perl, no se hace la distinción que se hacía en Pascal, entre funciones que retornan un valor y procedimientos, que no retornan un valor.

[2] Bueno, los puristas admitimos que las llaves son parte del bloque. Y Perl no requiere la identación del bloque, pero los nuevos programadores son estilizados

[3] A menos que tu subrutina valla a ser particularmente compleja y declare un prototipo, que indica como el compilador va a parsear e interpretar la invocación de los argumentos. Esto no es común, vea la página del manual `perlsub` para mas información.

La definición de la subrutina es global; sin hechiceria no hay subrutina privadas. Si tiene dos subrutinas definidas con el mismo nombre, la última subrutina sobre escribe la primera. Esto es generalmente considerado como una mala práctica, o una señal de que el programador de mantenimiento esta confundido.

Como pudo notarlo en el ejemplo previo, puedes usar variables globales dentro de el cuerpo de una subrutina. De hecho, todas las variables que hemos visto hasta ahora son globales; esto significa, que podemos acceder a ellas desde cualquier parte del programa. Esto horroriza a los puristas del lenguaje, pero no se preocupe, el equipo de desarrollo de Perl formo una turba iracunda con antorchas y corrió fuera de la ciudad hace años. Mas adelante veremos como hacer variables privadas en "Variables Privadas en Subrutinas".

4.2. Llamar una Subrutina

Puede invocar o llamar una subrutina desde cualquier expresión usando el nombre de la subrutina (con el signo &) [4].

```
1    &marino;    # dice: Hola, marinera numero 1!
2    &marino;    # dice: Hola, marinera numero 2!
3    &marino;    # dice: Hola, marinera numero 3!
4    &marino;    # dice: Hola, marinera numero 4!
```

4.3. Retornar Valores

Las subrutinas siempre son invocadas como parte de una expresión, incluso si el resultado de esa expresión no se usa para nada. Cuando llamamos anterior mente a la subrutina `&marino`, calculamos un valor en la expresión contenido en la llamada, pero el resultado simplemente se envió a la pantalla.

Muchas veces que se llama a una subrutina, se quiere hacer algo con el resultado. Entonces debe prestar atención al *valor de retorno* de una subrutina. Todas las subrutinas en Perl tienen un valor de retorno. No hay distinción entre las que retornan valores y las que no. Sin embargo no todas las subrutinas en Perl tiene un valor de retorno **útil**.

Puesto que en Perl, las subrutinas siempre deben devolver un valor, sería un poco inútil tener que declarar una sintaxis especial de tipo `return` para la mayoría de los casos. Entonces Larry hizo algo simple. Cualquier calculo que se encuentre de último en una subrutina es automáticamente el valor de retorno.

Por ejemplo, vamos a definir esta subrutina:

```
1    sub sum_of_fred_and_barney {
2        print "Hey,  has llamado a la subrutina sum_of_fred_and_barney !\n";
3        $fred + $barney; # Este es el valor de retorno.
4    }
```

La última expresión evaluada en el cuerpo de la subrutina, es la suma de $fred y $barney, entonces la suma de $fred y $barney va a ser el valor de retorno. Veamos esto en acción:

[4] Y frecuentemente un par de paréntesis, incluso si estan vacios

```
$fred = 3;
$barney = 4;
$wilma = &sum_of_fred_and_barney;   # $wilma obtiene 7
print "\$wilma es $wilma.\n";
$betty = 3 * &sum_of_fred_and_barney; # $betty obtiene 21
print "\$betty es $betty.\n";
```

Ahora supongamos que agrega una linea mas a la subrutina, por ejemplo:

```
sub sum_of_fred_and_barney {
    print "Hey,  has llamado a la subrutina sum_of_fred_and_barney !\n";
    $fred + $barney; # Este es el valor de retorno.
    print "Hey, estoy retornando un valor ahora!\n";
}
```

En este ejemplo, la última expresión evaluada no es una suma; es la sentencia print. Esta retorna normalmente 1, que significa "printing was successful", pero no es el valor de retorno que esperaba. Entonces, debe ser cuidadoso cuando agrega código adicional a una subrutina, puesto que la ultima expresión evaluada va a ser el valor de retorno.

Tome en cuenta que no es la última linea de la subrutina, es la última expresión evaluada. Por ejemplo, esta subrutina devuelve el valor mas grande, entre $fred y $barney:

```
sub larger_of_fred_or_barney  {
    if ($fred > $barney) {
        $fred;
    } else {
        $barney;
    }
}
```

La última expresión evaluada esta entre $fred o $barney, entonces el valor de una de estas variables va a ser el valor de retorno. No podemos saber cual va a ser la variable de retorno hasta que no veamos que guarda cada variable en tiempo de ejecución.

Estos son ejemplos triviales, se pone mejor cuando podemos pasar valores a la subrutina en lugar de depender de variables globales.

4.4. Argumentos

La subrutina llamada larger_of_fred_or_barney puede ser mas útil si no estuviera forzada a usar variables globales.

Perl tiene argumentos para subrutinas. Para pasar una lista de argumentos a una subrutina, simplemente se coloca la expresión de lista en paréntesis, después de la llamada a la subrutina, por ejemplo:

```
$n = &max(10, 15);  # Esta llamada tiene dos paréntesis.
```

Una vez que lista es enviada a la subrutina, va a estar disponible en la subrutina para hacer lo que sea necesario. Por supuesto, usted tiene que guardar esta lista en algún lugar, Perl automáticamente la lista de parámetros (este es otro nombre para la lista de argumentos) en una variable de tipo array especial llamada `@_` durante el tiempo de

existencia de la subrutina. La subruina puede acceder a esta variable para determinar o el número de argumentos y el valor de los argumentos.

Esto significa que el primer parámetro de la subrutina es guardado en `$_[0]`, el segundo es guardado en `$_[1]` y así sucesivamente. Pero, he aquí una nota importante, estas variables no tienen nada que ver con la variable `$_`, es solo que la lista de parámetros debe ser almacenada en una variable array para que la subrutina pueda utilizarlos, y Perl llama a esta variable `@_`.

Ahora usted puede escribir la subrutina &max para que se parezca un poco a la subrutina &larger_of_fred_or_barney, pero en lugar de usar $fred usted puede usar el primer parámetro de la subrutina ($_[0]), y en lugar de usar $barney, puede usar el segundo parámetro de la subrutina ($_[1]). Y así puede terminar un código que se vea como esto:

```
1    sub max {
2        # Compare esto con &larger_of_fred_or_barney
3        if ($_[0] > $_[1]) {
4            $_[0];
5        } else {
6            $_[1];
7        }
8    }
```

Bien, como ya dijimos, puede hacer esto. Pero es un poco feo con todos esos subíndices, y es poco fácil de escribir, revisar, y depurar. En un momento veremos una mejor forma de hacerlo.

Hay otro problema con esta subrutina, los parámetros adicionales son ignorados, puesto que la subrutina nunca mira en $_[2], a Perl no le importa si allí hay algo o no, y la ausencia de un parámetro también es ignorada, simplemente obtendrá `undef` si mira mas allá del final del array `@_`. En este mismo capítulo veremos como hacer esta subrutina mas eficiente.

La variable `@_` es privada para cada subrutina;<A menos que se coloque un signo & delante del nombre de la subrutina en la variable y no se indique ningún argumento o paréntesis, en cuyo caso `@_` es heredada desde el contexto de la llamada a la subrutina. Generalmente es una mala idea, pero en ocasiones es útil> si hay una variable global @_, su valor es conservado antes de la invocación de la subrutina, y su valor es restaurado cuando la llamada termina. Esto significa que una subrutina puede pasar valores a otra subrutina sin miedo a perder los valores contenidos en @_. Si una subrutina se llama recursivamente, cada invocación obtiene un nuevo `@_`, entonces `@_` es siempre la lista de parámetros para la subrutina.

4.5. Variables privadas en Subrutinas

Por defecto, todas las variables en Perl son variables globales; lo que significa que son accesibles desde cualquier parte del programa. Pero usted puede crear variables privadas llamadas variables léxicas, en cualquier momento usando el operador `my`. Por ejemplo:

```
1    sub max {
2        my ($m, $n);    # Nuevas variables privadas para este bloque
```

```
3         ($m, $n) = @_;  # obtener los parámetros
4         if ($m > $n) { $m } else { $n }
5     }
```

Estas variables son privadas (o de ámbito local) al bloque que las contiene, ningún otro código puede acceder o modificar estas variables privadas, por accidente o por diseño [5].

También es importante señalar que, en el bloque `if`, no fue necesario un punto y coma después de la expresión del valor de retorno. Aunque Perl permíte omitir el último punto y coma de un bloque, en la práctica tu puedes omitirlo solo cuando el código es tan simple que puedas escribirlo en un bloque de una sola linea.

La subrutina en el ejemplo previo puede ser mas simple aún. Debe haber notado que la lista ($m, $n) fue escrita dos veces, el operador `my` puede ser aplicado a una lista de variables encerradas en paréntesis, entonces se pueden combinar las dos primeras sentencias de la subrutina de la siguiente manera:

```
1     my($m, $n) = @_;    # nombre de los parámetros de la subrutina
```

Una sola sentencia crea las variables privadas y define sus valores. Casí todas las subrutinas van a comenzar con una linea muy parecida a esta, nombres entre paréntesis. Cuando vea esta linea, va a saber que la subrutina espera dos parámetros escalares, que vas a llamar $m y $n dentro de la subrutina.

4.6. Longitud de la lista de parámetros

En el mundo real, las subrutinas no poseen un valor arbritario para la longitud de la lista de parámetros. Esto es porque en Perl no hay límites innecesarios. Es agradable que Perl sea tan flexible, pero puede haber problemas cuando se llama a la subrutina con una cantidad diferente de parámetros que espera el autor.

Por su puesto, la subrutina puede fácilmente probar si el número de argumentos es correcto examinando el array @_. Por ejemplo, podemos escribir la función &max de cierta forma para que revise la lista de argumentos:

```
1     sub max {
2         if (@_ != 2){
3             print "WARNING! &max debería tener exactamente dos argumentos\n";
4         }
5         # ...
6     }
```

Perl en el mundo real, esto casi no se usa, es mejor hacer que la subrutina se adapte a los parámetros.

4.6.1. Subrutina &max bien escrita

```
1     $maximo = &max(4, 5, 10, 4, 6);
2
3     sub max {
```

[5] Programadores mas avanzados puede hacer una variable de ámbito local accesible por referencia desde fuera de su ámbito, pero nunca por el nombre de la variable

```
4           my ($max_so_far) = shift @_;    # tomo el primer valor como el mayor
5           foreach (@_) {
6               if ($_ > $max_so_far) {
7                   $max_so_far = $_;
8               }
9           }
10          $max_so_far;
11      }
```

4.7. Notas sobre Variables Léxicas (my)

Estas variables léxicas pueden usarse en cualquier bloque, no necesariamente en una subrutina. Por ejemplo, pueden usarse en un bloque de un `if`, `while`, o `foreach`:

```
1       foreach (1..10){
2           my ($cuadrado) = $_ * $_; # variable privada en este ciclo.
3           print "El cuadrado de $_ es $square.\n";
4       }
```

La variable $cuadrado es privada al bloque de código; en este caso, al bloque de código del ciclo foreach. Si no se encuentra encerrada en un bloque, la variable es privada para el archivo completo. El concepto importante es que el ámbito de una variable léxica es limitado al pequeño bloque de código que encierre la variable. Esto es una gran virtud para la mantenibilidad del código. Si hay un valor erróneo en $cuadrado, la culpabilidad va a estar limitada a una pequeña porción del código.

Debe haber notado, que el operador `my` tampoco altera el contexto de la asignación:

```
1       my ($num)   = @_; # contexto de lista.
2       my $num     = @_; # contexto escalar.
```

En la primera linea, $num obtiene el primer parámetro, en la segunda línea $num obtiene el número de parámetros en contexto escalar.

Vale la pena recordar, que el uso de `my` sin paréntesis solo va a declarar una variable léxica simple:

```
1       my $fred, $barney;      # MAL! Falla al declarar $barney
2       my ($fred, $barney);    # declara ambas variables.
```

También puede usar `my` para crear un nuevo y privado array:

```
1       my @phone_number;
```

4.8. Usando el Pragma Strict

Perl tiende a ser un lenguaje muy permisivo. Pero puede ser que quieras que Perl imponga un poco de disciplina; esto se puede hacer con el pragma `strict`.

Un pragma es una sugerencia al compilador, que le dice algo sobre el código. En este caso, el uso del pragma `strict` le dice al compilador interno de Perl que debe forzar el uso algunas buenas reglas de programación para el resto de este bloque o archivo.

¿ Porqué esto es importante ?, Bueno imagine que esta creando un programa y usted escribe una linea como esta:

```
1     $bamm_bamm = 3;  # Perl crea esta variable automáticamente
```

Ahora, luego de un rato, escribe un ciclo while. Después de que la linea anterior no sea visible en la pantalla, usted tipea una linea para incrementar la variable.

```
1     $bammbamm += 1;
```

Puesto que perl ve un nuevo nombre de variable (el guión bajo es significativo) crea una nueva variable, e incrementa su valor en uno. Si eres afortunado e inteligente, activaste las advertencias (warnings), y Perl te va a decir que has usado una o varias variables globales una sola vez en tu programa. Pero si eres simplemente inteligente, vas a usar cada variable mas de una vez, y Perl no va a advertirte nada.

Para decirle a Perl que quieres ser mas restrictivo, coloque `use strict` al principio de su programa.

```
1     use strict;     # Forza algunas buenas reglas de programación.
```

Ahora entre otras restricciones, Perl va a insistir que declares cada variable nueva, usualmente con `my`.

```
1     my $bamm_$bamm = 3;     # Nueva variable léxica.
```

Ahora, Perl va a notar que no hay una variable $bammbamm declarada, entonces tu error va ser automáticamente atrapado en tiempo de compilación.

```
1     $bammbamm += 1;     # No such variable: Compile time fatal error
```

Para aprender mas sobre las restricciones que aplica el pragma strict, le recomendamos ver la documentación oficial. La documentación de los pragmas se encuentra en un archivo bajo el nombre del pragma, entonces podemos usar `perldoc strict`.

La mayoría de la gente cree que si un programa es mas largo que la pantalla, generalmente necesita `use strict`. En esto estamos de acuerdo.

4.9. El operador return

El operador return inmediatamente retorna un valor de desde una subrutina:

```
1     my @names = qw/ fred barney betty dino wilma pebbles bamm-bamm /;
2     my $result = &which_element_is("dino", @names);
3
4     sub which_element_is {
5         my($what, @array) = @_;
6         foreach (0..$#array) {
7             if ($what eq $array[$_]) {
8                 return $_;
9             }
10        }
11        -1;
12    }
```

Esta subrutina es usada para encontrar el índice de “dino” en el array @names.

A algunos programadores les gusta usar return siempre, como una forma de documentar que este es el valor de retorno. Por ejemplo, se puede usar return cuando el valor de retorno no es la ultima expresión evaluada de la subrutina.

4.10. Omitir el signo &

Como lo prometimos, vamos a darle la regla para cuando una llamada a subrutina puede omitir el signo &. Si el compilador ve una definición de la subrutina antes de la llamada, o si Perl puede decidir que la sintaxis es una llamada a subrutina, la subrutina puede ser llamada sin un signo &. Como si fuera una función integrada de Perl. Pero hay una trampa escondida en esta norma, como veremos a continuación.

Esto significa que si Perl puede ver que es una llamada a una subrutina sin el signo &, por lo general va bien. Entonces si tienes una lista de parámetros entre paréntesis, entonces esto es una llamada a una función:

```
1    my $cards = shuffle(@deck_of_cards);  # & no es necesario.
```

Si el compilador interno de perl ya ha visto la definición de la subrutina, entonces generalmente funciona bien. En este caso particular, puedes incluso omitir los paréntesis que abrazan la lista de argumentos.

```
1    sub division {
2        $_[0] / $[1];
3    }
4
5    my $cociente = division 355, 113;   # Usa &division
```

Esto funciona por la regla de que los paréntesis pueden ser omitidos.

Pero no ponga la declaración de la subrutina despues de la llamada, o el compilador no sabrá de que se trata la llamada de `division`. Sin embargo esta no es la trampa, el problema en si es el siguiente: Si la subrutina tiene el mismo nombre que una función propia de Perl, usted *debe* usar el ampersand (&) para hacer la llamada a la subrutina. Al usar el ampersand, nos aseguramos de que estamos llamando a una subrutina; sin el; usted puede hacer la llamada solo si la subrutina no tiene el mismo nombre que una función propia de Perl. Por ejemplo:

```
1    sub chomp {
2        print "Mucho,  mucho!\n";
3    }
4
5    &chomp;     # El ampersand aquí no es opcional!
```

Sin el ampersand, estamos haciendo una llamada a la función `chomp` interna de Perl.

La regla general aquí es: Hasta que usted conozca los nombres de todas las funciones internas de Perl, siempre use ampersand en las llamadas a funciones.

En nuestro caso, podemos usar funciones en nuestra lengua materna (Español) para evitar caer en la trampa.

4.11. Retornando valores no escalares

Un escalar no es la única cosa que puede retornar una subrutina. Si llamas a tu subrutina en un contexto de lista [6], esta puede retornar una lista de valores.

[6] Puedes detectar en que contexto esta siendo llamada su subrutina, usando la función wantarray

Supongamos que quieres obtener un rango de números (como viene del operador de rango, ..), pero quieres habilitar el conteo hacia abajo. El operador de rango solo cuenta hacia arriba, pero es algo fácil de arreglar:

```
sub list_from_fred_to_barney {
    if ($fred < $barney){
        $fred..$barney;
    } else {
        reverse $barney..$fred;
    }
}

$fred = 11;
$barney = 6;

@c = &list_from_fred_to_barney;
```

Finalmente, puedes retornar nada. Cuando usamos `return` sin argumentos vamos a retornar `undef` en contexto escalar o una lista vacía en contexto de lista. Esto puede ser útil para retornar errores de una subrutina.

4.12. Variables Privadas Persistentes

Con `my` podemos hacer variables privadas en una subrutina, sin embargo cada vez que llamamos a la subrutina va a re-definir los valores una vez mas. Con `state`, podemos tener variables privadas del ambito de una subrutina pero Perl va a mantener sus valores entre las llamadas.

En el primer ejemplo de este capítulo, vimos una subrutina llamada marino, que incrementaba una variable:

```
sub marine {
    $n += 1;    # Variable global $n
    print "Hola,  marinera numero $n!\n"
}
```

Ahora que debemos usar `strict`, el uso de la variable gobal $n no esta permitido. No podemos hacer de $n una variable léxica con `my` porque entonces no va a mantener el valor.

Declarando nuestra variable con `state` le decimos a Perl que retenga el valor de esta variable entre las llamadas a la subrutina y hace que la variable sea una variable privada de la subrutina.

```
use 5.010;

sub marino {
    state $n = 0;   # privada, variable persistente $n
    $n += 1;
    print "Hola, marinera numero $n!\n";
}
```

Ahora, podemos obtener la misma salida mientras usamos `strict` sin usar variables globales. La primera vez que llamamos a la subrutina, Perl declara e inicializa `$n`, para las siguientes llamadas de la subrutina, Perl ignora la sentencia.

Podemos conservar el estado de cualquier variable, no es solo para los datos escalares. Aquí tenemos una subrutina que recuerda sus argumentos y provee una suma usando `state` en un array.

```
1   use 5.010;
2
3   running_sum(5, 6);
4   running_sum(1..3);
5   running_sum( 4 );
6
7   sub running_sum {
8       state $sum = 0;
9       state @numbers;
10
11      foreach my $number (@_){
12          push @numbers, $number;
13          $sum += $number;
14      }
15      say "La suma de (@numbers) es $sum";
16  }
```

La salida de este programa es:

Terminal

```
La suma de (5 6) es 11
La suma de (5 6 1 2 3) es 17
La suma de (5 6 1 2 3 4) es 21
```

Sin embargo, hay una ligera restricción en arrays y hashes como variables de estado. No podemos inicializarlas en contexto de lista. Por ejemplo:

```
1   state @array = qw(a b c);   # Error !
```

Esto nos dara un error que sugiere que podríamos usar esto en una versión futura de Perl:

Terminal

```
Initialization of state variables in list context currently forbidden
```

4.13. Ejercicios

1. Escriba una función llamada “total” que retorne el total de una lista de números. (Nota: la función no debe realizar algún tipo de I/O, esta debe simplemente procesar sus parámetros y retornar un valor). Complete el siguiente programa de ejemplo con la función “total”, el resultado de la sumatoria debe dar 25 para el primer grupo de números.

```
1       my @fred = qw{ 1 3 5 7 9 };
2       my $fred_total = total(@fred);
3       print "El total de \@fred es $fred_total.\n";
4       print "Ingrese algunos números separados por lineas: ";
```

```
5           my $user_total = total(<STDIN>);
6           print "El total para los números ingresados es: $user_total.\n";
```

2. Usando la función escrita en el ejercicio anterior, realice un programa que calcule la sumatoria de todos los números del 1 al 1000.

3. Escriba una función llamada &above_average que tome una lista de números y retorne solo aquellos números que estén por encima del promedio (Nota: escriba otra función que calcule el promedio de varios números dividiendo el total de la sumatoria de los números por el numero de items). Use su función para probar el siguiente programa:

```
1           my @fred = above_average(1..10);
2           print "\@fred es @fred\n";
3           print "(Debe ser 6 7 8 9 10)\n";
4           my @barney = above_average(100, 1..10);
5           print "\@barney es @barney\n";
6           print "(Debe ser solo 100)\n";
```

4. Escriba una función llamada "saludo", que de la bienvenida a una persona por su nombre y que ademas diga el nombre de la ultima persona saludada, ejemplo:

```
1           saludo("Fred");
2           saludo("Barney");
```

   Esto debe escribir lo siguiente:

```
1           Hola Fred! Eres el primero aquí!
2           Hola Barney! Fred también esta aquí!
```

5. Modifique el programa anterior para que imprima los nombres de todas las personas que han sido previamente saludadas.

```
1           saludo("Fred");
2           saludo("Barney");
3           saludo("Wilma");
4           saludo("Betty");
```

   La salida debe ser como lo siguiente:

```
1           Hola Fred! Eres el primero aquí!
2           Hola Barney! He visto a: Fred
3           Hola Wilma! He visto a: Fred Barney
4           Hola Betty! He visto a: Fred Barney Wilma
```

5 Entrada y Salida

Ya hemos visto antes como hacer alguna entrada/salida, con el fin de hacer algunos de los ejercicios. Pero ahora vamos a aprender más acerca de las operaciones que cubren el 80 % de la E/S que se necesita para la mayoría de los programas. Por ahora, sólo piense en "entrada estándar" como "el teclado", y "salida estándar" como "la pantalla".

5.1. Entrada desde la Entrada Estándar

Leer el flujo de datos que viene desde la entrada estándar es fácil. Ya hemos estado haciendo esto antes con el operador line-input STD. Si evaluamos este operador en contexto escalar, obtendremos la próxima linea de la entrada. Por ejemplo:

```
1     $line = <STDIN>;        # lectura de la siguiente línea
2     chomp($line);           # y chomp
3     chomp($line = <STDIN>); # lo mismo más idiomáticamente
4
```

Dado que del operador de línea de entrada devolverá `undef` cuando llegue al final del archivo, podemos usar esto para salir de un ciclo:

```
1     while (defined($line = <STDIN>)) {
2       print "I saw $line";
3     }
```

Muchas cosas ocurren esa primera línea: estamos leyendo la entrada en una variable, se comprueba que este definida, y si lo está (lo que significa que no hemos llegado al final de la entrada) entramos al bucle `while`. Luego dentro del bucle, vamos a tener cada linea una tras otra en la variable `$_`. Es importante resaltar que no estamos haciendo `chomp` de la entrada. En este tipo de bucles, no puedes poner `chomp` en la expresión condicional, entonces lo que se hace con frecuencia, si es necesario, es poner chomp en la primera linea del cuerpo del bucle. Como es algo que vas a querer hacer con bastante frecuencia, naturalmente como ya hemos visto antes, Perl tiene un atajo para hacer esto. El atajo se ve así:

```
1     while (<STDIN>){
2         print "Say: $_";
3     }
```

De esta manera, las lineas leídas de la entrada estándar, automáticamente van a estar contenidas en la variable mágica `$_` por defecto. Pero tenga cuidado, esto funciona solo cuando no hay mas nada que `<STDIN>` en la expresión condicional de un bucle `while`. Si colocas algo más en la expresión condicional, este atajo no va a funcionar.

Es importante destacar que cuando evaluamos el operador `line-input` en la expresión condicional de un ciclo while, lo estamos evaluando en un contexto escalar.

De otra manera, evaluando el operador `line-input` en contexto de lista, obtendremos todas las lineas de la entrada como una lista, cada elemento de la lista es una linea:

```
1    foreach (<STDIN>){
2        print "Say: $_";
3    }
```

No existe una conexión entre la variable mágica `$_` y el operador `line-input` (STDIN). Para este caso, la variable de control por defecto de un bucle `foreach` es `$_`.

Posiblemente no vea la diferencia aun entre como relaciona Perl al operador `line-input` y la variable mágica `$_` en el bucle `while` y el bucle `foreach`. En el bucle `while`, Perl lee la linea entrante y la asigna a la variable `$_`, y luego entra al bucle. Luego regresa, lee la siguiente linea y la asigna de igual manera a la variable mágica `$_` y entra nuevamente al bucle. Pero en un bucle `foreach`, el operador `line-input` es utilizado en un contexto de lista (porque el foreach necesita una lista que iterar). Entonces lee todas las lineas antes de iniciar el bucle. La diferencia la podemos ver en el momento en que tengamos que procesar por ejemplo 400 MB de archivos de registro (logs), en este caso generalmente es mejor usar `while`, pues va a procesar una linea a la vez.

El Operador Diamante (La Cometa)

Otra forma de leer la entrada de datos es con el operador diamante (o cometa): `<>`. Es muy útil para usar en programas que corren en sistemas operativos tipo Unix, con respecto a la invocación de argumentos. Si desea que sus programas en Perl puedan usarse como utilidades de Unix (cat, sed, sort, grep, lpr, y otras mas), el operador diamante va a ser su amigo.

La invocación de argumentos de un programa normalmente son un numero de "palabras" en la linea de comandos que vienen después del nombre del programa. En este caso, vamos a darle nombres de archivos y su programa va a procesarlos en secuencia:

Terminal

```
$ ./my_program fred barney betty
```

El comando anterior, correo el *comando* `my_program` (que debe estar en el directorio actual), y el programa va a procesar los archivos fred, barney, y betty

Si no invocamos argumentos, el programa va a procesar la entrada estándar. O, como caso especial, si le da un solo guion (-) como argumento esto también significa leer desde la entrada estándar. Entonces, si la lista de argumentos es: `fred - betty`, va a leer el archivo fred, luego va a leer la entrada estándar y luego va a leer el archivo betty.

La ventaja de hacer que sus programas funcionen de esta manera es que tu puedes escoger el tipo de entrada de datos en tiempo de ejecución, por ejemplo no necesitas reescribir un programa para usar una tubería (`|`). Larry puso esta característica en Perl, porque quería que fuera fácil para usted, escribir programas que trabajaran como utilidades estándar de Unix.

El operador diamante, es actualmente un operador especial de `line-input`. Pero en lugar de obtener la entrada desde el teclado, este toma los datos del tipo de entrada que el usuario escoja:

```
1     while (defined($line = <>)) {
2         chomp($line);
3         print "It was $line that I saw!\n";
4     }
```

Entonces, con este programa, si le damos los archivos fred, barney, y betty, va a procesar las lineas del archivo fred en la variable $line, luego que termine con el archivo fred va por el archivo barney y luego por el archivo betty. Note que el ciclo no se rompe entre un archivo y otro, cuando usamos el operador diamante, todos los archivos se juntan en un solo archivo grande.

El operador diamante va a retornar `undef` cuando llegue al final de todo el conjunto de archivos, con lo que se va a terminar el ciclo.

De igual manera el atajo de la variable mágica `$_` se puede usar con este operador. Por ejemplo:

```
1     while(<>){
2         chomp;
3         print Ït was $_ that I saw!\n";
4     }
```

Esto funciona exactamente igual al código anterior, solo que escribimos menos. Y debe haber notado que estamos usando el comportamiento por defecto de `chomp`; si no le damos argumentos a `chomp`, va a trabajar con la variable mágica `$_`.

Cuando el operador de diamante no puede abrir un archivo, muestra un mensaje de diagnostico:

Terminal

```
Can't open maria: No such file or directory
```

5.2. Invocación de Argumentos

Técnicamente, usar el operador de diamante no hace invocación de argumentos. El funciona usando el array `@ARGV`. Este array es un array especial que esta predefinido en el interprete de Perl como la lista de invocación de argumentos. En otras palabras, este es otro array con un divertido nombre en mayúsculas sostenidas, cuando tu programa inicia, `@ARGV` ya esta lleno con la lista de argumentos.

Puedes usar el array `@ARGV` como cualquier otro array, puedes sacar cosas de el con `shift` o usar un `foreach` para iterarlo.

El operador de diamante busca en @ARGV para determinar que nombres de archivos debe usar. Si encuentra una lista vacía, usa lo que viene de la entrada estándar. Esto significa que después que el programa inicia y antes de usar el operador de diamante, tienes chance de jugar con `@ARGV`. Por ejemplo, si sabemos que archivos queremos procesar, podemos poner los nombres en `@ARGV` sin necesidad de leerlos de la entrada estándar.

```
1     @ARGV = qw# larry moe curly #;  # Forza leer estos archivos
2     while (<>){
3         chomp;
4         print "$_ Esto fue lo que vi en los archivos títeres";
5     }
6 =end programlisting
```

5.3. Salida hacia la Salida Estándar

El operador print, toma un lista de valores y envía cada elemento (como una cadena por supuesto) hacia la salida estándar, uno después del otro. No agrega nada delante, atrás, o entre los elementos. Si quiere algo adicional, como un salto de linea o un espacio, tiene que colocarlo a mano usted mismo.

Por su puesto, esto significa que hay una diferencia entre imprimir un array e interpolar un array:

```
1     print @array;   # Imprime la lista de elementos
2     print "@array"; # Imprime una cadena (el array interpolado)
```

La primera instrucción va a imprimir la lista de elementos, uno tras el otro, sin colocar espacios en el medio ni nada. La segunda instrucción va a imprimir cada elemento con un espacio entre cada elemento. Ejemplo.

Terminal

```
DB<36> @a = qw/maria pedro juan/

DB<38> print @a                                                    mariape

DB<39> print "@a";
  maria pedro juan
```

Dado que `print` espera una lista de cadenas y las imprime, sus argumentos son evaluados en contexto de lista. Dado que el operador diamante devuelve una lista de lineas en contexto de lista. Podemos reescribir las dos clásicas herramientas de Unix (**cat** y **sort**).

```
1     print <>;           # cat en Perl
2
3     print sort <>;      # sort en Perl
```

Claro, para ser honestos, los comandos `cat` y `sort` tienen una serie de funcionalidades adicionales que estos pequeños programas no tienen.

Seguro a notado que `print` al ser una función, puede omitir los paréntesis, recuerde que puede hacer esto siempre y cuando quitar los paréntesis no cambie el significado de la expresión, supongamos que tenemos el siguiente ejemplo:

```
1     print (2+3);
```

Parece una llamada a función y es una llamada a función. Esto imprime 5, y retorna un valor como cualquier otra función. El valor de retorno de `print` es verdadero o falso, indicando que efectivamente realizo la impresión, a menos que ocurra algún error del tipo I/O, el resultado normal de `print` va a ser **1**.

```
1     $result = print("Hola mundo\n");
```

Pero supongamos que tenemos el siguiente código:

```
1     print (2+3)*4;
```

¿ Que cree que va a ocurrir con esta instrucción ?, definitivamente no va a salir 20 por la pantalla. Perl va a imprimir 5, y va a multiplicar 1 por 4, y el resultado lo va a tirar a la basura.

Este es un problema típico de los paréntesis opcionales, a veces los humanos olvidamos que los paréntesis realmente importan. Cuando usamos print sin los paréntesis, print es

un operador de lista, imprime todos los elementos de esa lista. Pero cuando la primera cosa después de print es un paréntesis, print es una llamada de función, y va a imprimir solo lo que encuentre entre los paréntesis.

Actualmente, esta regla "Si parece una llamada a una función, es una llamada a función, aplica para todas las funciones de lista.

5.4. Formatear la salida con printf

Puede ser que quiera tener mas control con la salida de lo que ofrece print. En efecto, puede estar acostumbrada a formatear la salida con la función `printf` de Perl. No se preocupe, Perl provee una operación compatible con el mismo nombre.

El operador `printf` toma una cadena con formato seguida de una lista de cosas a imprimir. Ejemplo:

```
1    printf "Hola, %s; su contraseña expira en %d días!\n",
2    $user, $dias_para_morir;
```

La cadena con formato posee un numero de conversiones, estas conversiones comienzan con un signo de porcentaje (%) y termina con una letra. Debe haber la misma cantidad de elementos en la cadena con formato que en la lista que se pasa como argumento.

Para imprimir números generalmente es bueno usar la conversión `%g`, que automáticamente escoje entre punto flotante, entero, o notación exponencial, cual sea necesaria.

```
1    printf "%g %g %g\n", 5/2, 51/17, 51 ** 17; # 2.5 3 1.0683e+29
```

El formato `%d` indica decimal entero truncado.

```
1    printf "in %d days!\n", 17.85; # in 17 days!
```

Note que este proceso de truncado no redondea, en un momento vamos a ver como redondear números.

En Perl, `printf` es usado con frecuencia para datos en columnas, la mayoría de los formatos permiten indicar el ancho del campo. Si los datos no llenan el campo, de igual manera el campo se expande dejando del espacio requerido.

```
1    printf "%6d\n", 42; # se vería asi:  ````42 (` representa espacio blanco)
2    printf "%2d\n", 2e3 + 1.95; # 2001
```

La conversión `%s` se usa para las cadenas, efectivamente interpola el valor obtenido como una cadena, y también se le puede indicar el tamaño del campo:

```
1    printf "%10s\n", "wilma"; # se vería `````wilma
```

Un valor negativo en el tamaño del campo, indica justificado a la izquierda:

```
1    printf "%-15s\n", "flinstone";  # se ve flinstone`````
```

La conversión `%f` redondea, y permite indicar cuantos números quiere después del punto decimal.

```
1    printf "%12f\n", 6 * 7 + 2/3;   # se ve ```42.666667
2    printf "%12.3f\n", 6 * 7 + 2/3; # se ve ``````42.667
3    printf "%12.0f\n", 6 * 7 + 2/3; # se ve ``````````43
```

Para imprimir un signo de porcentaje, usamos `%%`:

```
1    printf "Interés Mensual : %.2f%%\n",  5.25/12; # El valor seria "0.44%
```

5.5. Arrays y printf

Generalmente, usted no va a usar un array como argumento para `printf`. Esto es porque un array almacena cualquier numero de elementos, y la cadena de formato viene dada para trabajar con un numero finito de elementos. Si hay tres conversiones de formato, deben haber exactamente tres elementos.

Pero no hay razón para no hacer la cadena de formato al vuelo, puesto que puede ser cualquier expresión. Este puede ser un truco para hacer esto:

```
1    my @items = qw( wilma dino pebbles );
2    my $format = "The items are:\n" . ("%10s\n" x @items);
3    ## print "the format is >>$format<<\n"; # for debugging
4    printf $format, @items;
```

Aquí usamos el operador `x` (Que aprendimos en el capítulo 2) para replicar la cadena dada el número de veces dado por `@items` (en contexto escalar).

Pero podemos hacer esto mas corto y mas mágico:

```
1    printf "The items are:\n".("%10s\n" x @items), @items;
```

Note que @items esta siendo usado en contexto escalar, para obtener su longitud, y en contexto de lista para obtener sus valores. El **Contexto** es importante.

5.6. Filehandles

Un Filehandle es el nombre en un programa en Perl para una conexión de E/S entre el proceso Perl y el mundo exterior. Es decir, es el nombre de una conexión, no necesariamente el nombre de un archivo.

Los Filehandle se llaman igual que otros identificadores de Perl (con letras, dígitos y pisos bajos) pero no pueden comenzar con un dígito.

Actualmente hay seis nombres de Filehandle especiales en Perl que se usan para propósitos específicos: STDIN, STDOUT, STDERR, DATA, ARGV, y ARGVOUT. Aunque puede elegir cualquier nombre para un Filehandle, no debería usar ninguno de estos seis nombres, a menos que quiera hacer algo muy especial. [1].

Cuando su programa inicia, el Filehandle llamado `STDIN` es la conexión entre el proceso Perl y lo que sea que el programa tenga como entrada, conocido como entrada estándar o *standard input stream*. Esto generalmente es el teclado del usuario a menos que el usuario use otra cosa como fuente de datos para la entrada, como un archivo o la salida de otro programa a través de una *tubería*. También esta la salida estándar o *standard output stream*, que es es Filehandle `STDOUT`. Por defecto, esta conectada con la pantalla del usuario, pero el usuario puede enviar la salida a un archivo o a otro programa a través de una *tubería*. Esto del flujo estándar o *standard stream* viene de la librería de Unix *standard I/O*.

La idea general es que el programa lea desde `STDIN` y escriba en `STDOUT`, confiando en el usuario (o generalmente un programa que inicie su programa). De esta forma, el usuario puede escribir una linea de comandos en el prompt del shell como esta:

[1] En algunos casos puede reusar estos nombres sin ningún problema. Pero un programador menos experimentado que mantenga el código va a confundirse

Terminal
```
$ ./su_programa <dino >willma
```

Esto le dice a la terminal, que la entrada de su programa viene de *dino* y la salida va a *willma*. Esto funciona siempre y cuando la entrada de su programa sea STDIN y la salida valla a STDOUT.

Sin ninguna modificación el programa va a trabajar sin problemas con *tuberías*. Este es otro concepto de Unix, que le permite escribir lineas de comando como la siguiente:

Terminal
```
$ cat fred barney | sort | ./su_programa | grep algo | lpr
```

5.6.1. Abrir un Filehandle

Como ya sabe, Perl provee tres filehandles - STDIN, STDOUT, STDERR, que automáticamente abren archivos o dispositivos establecidos por el programa padre del proceso (probablemente la terminal). Cuando necesita otros filehandles, debe usar el operador `open` para decirle a Perl, que le pida al sistema operativo que abra una conexión entre su programa y el mundo exterior. Aquí tenemos algunos ejemplos:

```
1  open CONFIG, "dino";
2  open CONFIG, "<dino";
3  open BEDROCK, ">fred";
4  open LOG, ">>logfile";
```

El primero, abre un filehandles llamado `CONFIG` a un archivo llamado `dino`. Esto es, el archivo *dino* va a ser abierto y lo que sea que tenga en su contenido, va a estar disponible en el programa a través del filehandle llamado `CONFIG`. El segundo hace exactamente lo mismo que el primero, pero con un signo *menor que* indicando "Use este nombre de archivo como entrada", a pesar de que eso es lo que hace por defecto.

Para abrir el filehandle BEDROCK para enviar la salida allí, usamos el signo *mayor que*, esto envía la salida al archivo fred. Si el archivo ya existe y tiene contenido, automáticamente reemplaza el contenido del archivo con la salida que enviamos al filehandle.

El cuarto ejemplo, indica como podemos usar dos signos *mayor que* para abrir un archivo sin reemplazar su contenido y anexar nuestra salida al final del archivo. Si no existe el archivo, lo crea y agrega el contenido, como si se tratara de un solo signo *mayor que*. Esta es una forma de manejar los archivos de registro (logfile). El programa puede escribir un grupo de lineas al archivo de registro cada ves que se ejecuta.

Puedes usar una expresión en lugar de un nombre de filehandle específico. Luego solo queda indicar la dirección explícita:

```
1  my $selected_output = "my_output";
2  open LOG,  "> $selected_output";
```

Note que el espacio después del signo *mayor que*. Perl ignora esto.

En desde la version 5.6 de Perl, puedes usar un tercer argumento extra para `open`:

```
1  open CONFIG, "<", "dino";
2  open BEDROCK, ">", $file_name;
3  open LOG, ">>", &logfile_name();
```

5.6.2. Filehandles Malos

Perl actualmente no abre el archivo por el mismo, Perl le pide al sistema operativo que abra el archivo por el. Por su puesto, el sistema operativo puede negarse a hacerlo debido a una configuración de permisos, un nombre de archivo incorrecto u otras razones.

Si intentas leer de un filehandle malo, vas a ver inmediatamente un `end-of-file` o fin de archivo. `end-of-file` indica `undef` en contexto escalar o una lista vacia en contexto de lista.

Si intentas escribir en un filehandle malo, los datos simplemente se perderan.

Afortunadamente, estas consecuencias son fáciles de evitar. Primero que todo, si estamos usando el pragma `warnings` o el switch `-w`, Perl va a advertirnos cuando vea que estamos usando un filehandle malo. Pero incluso antes de esto, la llamada `open` siempre nos indica si puedo hacer el trabajo o fallo, retornando verdadero o falso. Entonces, puedes escribir tu código de la siguiente manera:

```
1     my $success = open LOG, ">>logfile"; # capture the return value
2     if ( ! $success) {
3         # Fallo Open
4         ...
5     }
```

5.6.3. Cerrar un Filehandle

Cuando termines de trabajar con el Filehandle, puedes cerrarlo con el operador de cierre, de la siguiente forma:

```
1     close BEDROCK;
```

Cerrar un Filehandle le dice al sistema operativo, que hemos terminado la conexión con el archivo, entonces los últimos datos enviados por esa conexión son escritos al disco, si eso es lo que se espera.

Perl, automáticamente cierra un filehandle si lo abres de nuevo (esto ocurre si reusas el nombre del filehandle en otra llamada a `open`) o si sales del programa.

Debido a esto, varios programas simples de Perl no cierran los filehandles. Pero si quieres ser ordenado, debe haber una llamada a `close` por cada llamada a `open`. En general, es mejor cerrar cada filehandle pronto despues de haber terminado con el. Aun que este cerca del fin del programa.

5.7. Errores fatales con die

Miremos hacia un lado un momento. Necesitamos saber algunas cosas mas que aunque no estan relacionadas con los procesos de I/O nos va a permitir salir de un programa antes de lo esperado.

Cuando ocurre un error fatal en Perl (por ejemplo, si haces una division por cero, uso de una expresión regular inválida, o llamar una subrutina que no esta definida), sus

programas se rompen con un mensaje de error indicando la razón del porqué [2]. Esta funcionalidad esta disponible para nosotros a través de la función `die`, entonces podemos hacer nuestros propios errores fatales.

La función `die` imprime el mensaje que le pasas como argumento, y se asegura que el programa termine con una salida diferente de cero.

Probablemente no lo sepa, pero cada programa que se corre en Unix tiene un estado de salida, indicando cuando termina satisfactoriamente y cuando no. Programas que corren otros programas (como la utilidad make) mira el estado de salida para saber si todo va bien. El estado de salida es un simple byte, tradicionalmente 0 es *termine satisfactoriamente* y diferente de 0 es lo contrario.

Entonces, podemos escribir el ejemplo anterior de la siguiente manera:

```
1    if ( ! open LOG, ">>logfile") {
2        die "No se pudo crear el archivo de logs: $!";
3    }
```

Si `open` falla, `die` va a terminar el programa diciendo: "No se pudo crear el archivo de logs. Pero, ¿ Que es esa variable $! en el mensaje ?, Esto es, una queja del sistema que un ser humano puede leer. En general, cuando el sistema se niega a hacer algo que le estamos pidendo, en `$!` esta la razon del porqué.

Hay otra cosa interesante que `die` puede hacer por nosotros, automáticamente va a indicar el nombre del programa y el número de linea en donde ocurrio el error.

Suponiendo que en el programa anterior no podemos abrir el filehandle por problemas con los permisos, vamos a obtener el siguiente mensaje:

Terminal

```
No se pudo crear el archivo de logs: permission denied at your_program line 1234.
```

5.8. Mensajes de Advertencia con warn

Así como `die` indica un mensaje de error fatal, puedes usar `warn` para indicar un advertencia como si fuera una advertencia interna de Perl.

La función `warn` trabaja igual que la función `die`, excepto que esta función no termina el programa. Pero si agrega el nombre del programa y la linea e imprime el mensaje hacia la salida estándar de errores (STDERR).

5.9. Usando los Filehandle

Cuando un filehandle se abre para leer, puedes leer las lineas justo como leemos la entrada estándar con STDIN. Entonces, por ejemplo para leer las lineas de un archivo passwd de Unix:

```
1    if ( ! open PASSWD, "/etc/passwd") {
2        die "How did you get logged in? ($!)";
3    }
```

[2] Bueno, esto pasa por defecto, pero podemos atrapar los errores con la función eval, que vamos a ver mas adelante en otro capítulo

```
4
5       while (<PASSWD>) {
6           chomp;
7       }
```

Un filehandle abierto para escritura puede usarse con `print` o `printf`, colocandolo inmediatamente despues de la palabra reservada pero antes de la lista de argumentos:

```
1       print LOG "Captain's log, stardate 3.14159\n"; # output goes to LOG
2       printf STDERR "%d percent complete.\n", $done/$total * 100;
```

5.9.1. Cambiar un Filehandle de salida por defecto

Por defecto, si no le indicas un filehandle a `print` o a `printf`, la salida va a ir a STDOUT. Pero esto se puede cambiar con el operador `select`. Por ejemplo, vamos a enviar un par de lineas al filehandle BEDROCK:

```
1       select BEDROCK;
2
3       print "Hola esto es una prueba de piedra, o una piedra de prueba";
4       print "Willma!\n";
```

Hay que tener cuidado con esto, una vez que se termine de usar, se debe volver al valor por defecto STDOUT. De igualmanera, por defecto, la salida hacia un filehandle se guarda en el buffer. Configurando la variable especial `$|` a 1, va a hacer que el filehandle selecionado siempre haga flush del buffer despues de cada operación de salida. Entonces si se quiere asegurar que los archivos de log tengan los registros a tiempo, en el caso que quieras estar lellendo el log para monitorear el progreso del programa, puedes hacer algo como esto:

```
1       select LOG;
2       $| = 1;         # No mantenga las entradas de LOG en el buffer.
3       select STDOUT;
4       # El tiempo pasa, los niños aprenden a caminar, Chavez no se va  ...
5       print LOG "Esto es escrito en el log de una vez\n";
```

5.9.2. Reabrir un Filehandle por defecto

Ya mencionamos antes que si abres un archivo el anterior se cierra automáticamente. Y tambien digimos que no debería reusar uno de los seis nombres de filehandles por defecto, al menos que quiera hacer algo especial. Adicionalmente digimos que los mensajes de error en Perl van a STDERR. Si usa estas tres piezas de información juntas, ahora puedes tener una idea de como enviar los mensajes de error a un archivo.

```
1       # Envia los errores a mi log privado de errores
2       if ( ! open STDERR, ">>/home/elsanto/.error_log"){
3           die "No pude abrir el log de errores: $!";
4       }
```

¿ Que pasa si no es posible abrir el archivo de LOGS ?, la operación de reapertura de uno de los tres filehandles (STDIN, STDOUT, STDERR) falla, Perl amablemente restaura al valor original.

5.10. Salida con say

Perl 5.10 toma la función `say` del desarrollo de Perl 6. Es lo mismo que `print`, con la diferencia de que agrega al final un caracter de nueva linea (`\n`). Las siguiente sentencias producen la misma salida:

```
1    use 5.010;
2
3    print "Hello!\n";
4    print "Hello!", "\n";
5    say "Hello!";
```

Para interpolar una variable escalar o una lista, debe colocarse la variable entre comillas dobles, ejemplo:

```
1    use 5.010;
2
3    my @array = qw( a b c d );
4
5    say @array;     # "abcd\n"
6
7    say "@array";   # "a b c d\n"
```

Al igual que con `print`, puedes especificar un filehandle con `say`:

```
1    use 5.010;
2
3    say BEDROCK "Hola !";
```

Esta es una nueva caracteristica de Perl 5.10, y nosotros la usamos solo cuando estamos usando alguna otra función de Perl 5.10. La vieja y confiable `print` sigue siendo tan buena como siempre lo ha sido, pero sospechamos que hay algunos programadores de Perl que van a querer ahorrase el tener que escribir cuatro caracteres extra. (dos del nombre y \n).