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. 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. 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.

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)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, va a conformar el cuerpo de la subrutina, quedaría algo como esto:

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

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 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".

Llamar una Subrutina

Puede invocar o llamar una subrutina desde cualquier expresión usando el nombre de la subrutina (con el signo &) Y frecuentemente un par de paréntesis, incluso si estan vacios.

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

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:

```
    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.
    }
```

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:

```
    $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.

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:

```
sub max {
    # Compare esto con &larger_of_fred_or_barney
    if ($_[0] > $_[1]) {
        $_[0];
    } else {
        $_[1];
    }
}
```

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.

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:

```
sub max {
    my ($m, $n);    # Nuevas variables privadas para este bloque
    ($m, $n) = @_;  # obtener los parámetros
    if ($m > $n) { $m } else { $n }
}
```

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 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.

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:

```
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.

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:

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

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

Subrutina &max bien escrita

```
    $maximo = &max(4, 5, 10, 4, 6);

    sub max {
        my ($max_so_far) = shift @_;    # tomo el primer valor como el
mayor
        foreach (@_) {
            if ($_ > $max_so_far) {
                $max_so_far = $_;
            }
        }
        $max_so_far;
    }
```

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`:

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

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:

```
    my ($num)   = @_; # contexto de lista.
    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:

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

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

```
    my @phone_number;
```

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:

```
$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.

```
$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.

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

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

```
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.

```
$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.

El operador return

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

```
my @names = qw/ fred barney betty dino wilma pebbles bamm-bamm /;
my $result = &which_element_is("dino", @names);

sub which_element_is {
    my($what, @array) = @_;
    foreach (0..$#array) {
        if ($what eq $array[$_]) {
            return $_;
        }
    }
    -1;
}
```

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.

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:

```
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.

```
sub division {
    $_[0] / $[1];
}

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:

```
sub chomp {
    print "Mucho,  mucho!\n";
}

&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.

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 Puedes detectar en que contexto esta siendo llamada su subrutina, usando la función wantarray, esta puede retornar una lista de valores.

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.

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.

```
use 5.010;

running_sum(5, 6);
running_sum(1..3);
running_sum( 4 );
```

```
sub running_sum {
    state $sum = 0;
    state @numbers;

    foreach my $number (@_){
        push @numbers, $number;
        $sum += $number;
    }
    say "La suma de (@numbers) es $sum";
}
```

La salida de este programa es:

```
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:

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

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

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

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.

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

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

```
saludo("Fred");
```

```
saludo("Barney");
```

Esto debe escribir lo siguiente:

```
Hola Fred! Eres el primero aquí!
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.

```
saludo("Fred");
saludo("Barney");
saludo("Wilma");
saludo("Betty");
```

La salida debe ser como lo siguiente:

```
Hola Fred! Eres el primero aquí!
Hola Barney! He visto a: Fred
Hola Wilma! He visto a: Fred Barney
Hola Betty! He visto a: Fred Barney Wilma
```