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".

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:

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

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

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

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:

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

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:

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

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:

```
$ ./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:

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

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:

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

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:

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

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.

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

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:

```
    print @array;   # Imprime la lista de elementos
    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.

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


  DB<38> print @a

mariapedrojuan


  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**).

```
    print <>;           # cat en Perl


    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:

```
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**.

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

Pero supongamos que tenemos el siguiente código:

```
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.

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:

```
printf "Hola, %s; su contraseña expira en %d días!\n",
$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.

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

El formato %d indica decimal entero truncado.

```
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.

```
    printf "%6d\n", 42; # se vería asi:  ````42 (` representa espacio
blanco)
    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:

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

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

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

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

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

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

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

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:

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

```
    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.

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

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:

```
$ ./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:

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

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:

```
open CONFIG, "dino";
open CONFIG, "<dino";
open BEDROCK, ">fred";
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:

```
my $selected_output = "my_output";
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`:

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

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:

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

Cerrar un Filehandle

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

```
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.

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

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

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:

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

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).

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:

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

```
while (<PASSWD>) {
    chomp;
}
```

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:

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

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:

```
select BEDROCK;

print "Hola esto es una prueba de piedra, o una piedra de prueba";
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:

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

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.

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

¿ 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.

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:

```
use 5.010;

print "Hello!\n";
print "Hello!", "\n";
say "Hello!";
```

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

```
use 5.010;

my @array = qw( a b c d );

say @array;     # "abcd\n"

say "@array";   # "a b c d\n"
```

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

```
use 5.010;

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).