programa de BECAS
DGTIC
de FORMACIÓN en TIC

Seguridad Informática

Programación con Perl

Cervantes Varela Juan Manuel
Vallejo Fernández Rafael Alejandro

Funciones

- **chop**

chop($string)

Elimina el último caracter de una cadena de texto.

Actualiza el valor de $string.

Ejemplo:

```
$cadena = “perla”; # perla
chop($cadena); # devuelve: perl
```

- **chomp**

chomp($string)

Elimina el último caracter de la variable SOLAMENTE SI ES un salto de línea -> “\n”

Actualiza el valor de $string.

```
1  $cadena = "hola mundo!\n\n";
2  chomp $cadena;      # "hola mundo!\n"
3  chomp $cadena;      # "hola mundo!"
4  chomp $cadena;      # "hola mundo!"
5  chomp $cadena;      # "hola mundo!"
```

Funciones

- **length**

length(string)

Indica la cantidad de caracteres en una cadena.

Ejemplo:

```
$cad = "perl";
$tam = length($cad);  # $tam = 4
print "Tam = $tam";  # Tam = 4;
```

Funciones

- **index**

index(cadena, subcadena [, posInicio])

Busca en **cadena** la primera ocurrencia de la **subcadena** en o antes de **posInicio** (por defecto y negativos es '0').

Devuelve: posición del primer caracter de **subcadena** ó -1 si no la encuentra.

Ejemplo:

```
$string = "primer aparicion de la cadena hola hola";
$substring = "hola";
$posInic = 0;
index($string, $substring, $posInic);  # es 31
```

Funciones

- **rindex**

rindex(cadena, subcadena[, posInicio]);

Busca en **cadena** la última aparición de la **subcadena** en o antes de **posInicio** (por defecto length(cadena) y '0' si son negativos).

Devuelve: posición del primer caracter de **subcadena** ó -1 si no la encuentra.

Ejemplo:

```
$string = "ultima aparicion de la cadena hola hola";
$substring = "hola";
rindex($string, $substring);  # es 35
```

Funciones

- **substr**

Obtiene una subcadena de la cadena que recibe.

substr(cadena, índice, tamaño);

El índice puede ser negativo (lo recorre desde el final de la cadena)

Ejemplo:

```
$string = "prueba de substr";
$pos = 7;  # índice de letra 'd'
$tam = 2;  # tam de palabra 'de'
$extraer = substr($string, $pos, $tam) ;  # $extraer = "de"
```

Funciones

- **substr**

También puede actualizar la cadena reemplazando una subcadena (a partir de index y tam) por una nueva.

Ejemplo:

```
$string = "prueba de substr";
substr($string, 0, length("prueba")) = "poc";  # $string = "poc de substr"
print "$string";  # "poc de substr"
```

Funciones

- **uc(string)**

Devuelve toda la cadena en mayúsculas.

Ejemplo:

```
uc("hola");  # HOLA

uc('gRIto');  # GRITO
```

- **lc(string)**

Devuelve toda la cadena en minúsculas.

```
lc('ADIOS');  # adios

lc('oTRa');  # otra
```

Funciones

- **ucfirst(string)**

Devuelve la cadena con la primer letra en mayúsculas (sin alterar el resto).

Ejemplo:

```
ucfirst("hola");  # Hola

ucfirst('gRIto');  # GRIto
```

- **lcfirst(string)**

Devuelve la cadena con la primer letra en minúsculas (sin alterar el resto).

```
lcfirst('ADIOS');  # aDIOS

lcfirst('oTRa');  # oTRa
```

Funciones

- **split**

Separa una **cadena** en **subcadenas** de acuerdo al **patrón** que se indique.

split(/patrón/, cadena[, límite]);

límite: n-1 subcadenas que separará

/patrón/: expresión regular, si no se indica será **/\s+/** por defecto.

Devuelve: arreglo con las subcadenas resultantes.

Ejemplo:

```
split(/\t+/,"test\tsepara\t\tstrings\t");
# devuelve el arreglo: ('test', 'separa', 'strings')
```

Funciones

- **join**

Une una **lista de cadenas** (separadas por comas o con qw()) usando el primer argumento como **separador**.

join(separador, cadena[,cadena,…]);

join(separador, qw(cadena cadena ...));

Devuelve: cadena unida

Ejemplo:

```
join('-','une', 'cadenas', 'o', 'strings');  # devuelve: une-cadenas-o-strings
join("+", qw(une cadenas o strings)); # devuelve: une+cadenas+o+strings
```

Funciones

- **reverse**

Recibe un arreglo y lo devuelve con el orden invertido. Es decir, el último elemento es el primero.

reverse(@arreglo)

Ejemplo:

```
my @invertido = reverse(1,2,3,4,5,6,7);
# @invertido = (7,6,5,4,3,2,1);

my @arreglo = reverse(qw(uno dos tres cuatro));
# @invertido = ("cuatro", "tres", "dos", "uno");
```

Funciones

- **sort**

Recibe un arreglo y lo devuelve ordenado alfabéticamente de forma ascendente (sin modificar al original).

sort(@arreglo)

Ejemplo:

```
1  @desordenado = qw(lapiz pluma casa mueble mesa puerta)
2  @ordenado = sort(@ordenado);
3  # @ordenado = ('casa', 'lapiz', 'mesa', 'mueble', 'pluma', 'puerta')
```

Funciones

- **sort**

También puede tomar un bloque de código entre '{}' que indica como ordenar los elementos mediante la comparación de 2 variables globales, $a y $b.

sort {$a opComp $b} (@arreglo);

opComp es el operador de comparación

Ejemplo:

```
1  sort {$a <=> $b} (@array);  # ordena numericamente (ascendente)
2  sort {$b <=> $a} (@array);  # ordena numericamente (descendente)
3  sort {$b cmp $a} @array;  # ordena alfabéticamente (descendente)
```

Funciones matemáticas

```php
abs($num);  # devuelve: valor absoluto de un número

sin($radianes);  # devuelve: seno de un valor dado en radianes
cos($radianes);  # devuelve: coseno de un valor dado en radianes

$num ** $potencia;  # el operador ** eleva el $num a la $potencia

sqrt($num);  # devuelve: raíz cuadrada de $num

exp($num);  # devuelve: e (numero de euler) a la potencia $num
log($num);  # devuelve: logaritmo natural (base e) de $num
```

Expresiones regulares

- Para buscar patrones en cadenas de texto se utilizan los caracteres '=~' y el par de delimitadores del patrón. Por defecto son '/' '/'.

```
$str =~ /patronAbuscar/;
# devuelve:
# 1 [verdadero]: si el patron aparece en $str
# "" [falso]: si no encuentra el patron
```

- Ejemplo:

```
"palabra" =~ /ala/;  # => 1 [verdadero]
"palabra" =~ /ola/;  # => "" [falso]
```

Expresiones regulares

- Para cambiar los delimitadores por defecto, se utiliza la letra 'm' [match] antes del delimitador que se elija.

```
$str =~ m#patronAbuscar#;
$str =~ m"patronAbuscar";
$str =~ m%patronAbuscar%;
```

* Si se utilizan las comillas simples o '$' como delimitador, se pierde la interpolación.

Expresiones regulares

- Caracteres especiales para expresiones regulares

Caracter especial	Significado
.	Cualquier carácter excepto "\n"
\w	Palabra con letras o dígitos [a-zA-Z0-9]
\W	Cualquier cosa que no sea palabra [digito o letra]
\s	Espacio en blanco: [\n\r\t\f]
\S	Cualquier cosa que no sea "espacio en blanco"
\t,\n,\r	Tabulador, salto de línea, retorno de carro
\d	Dígito decimal [0-9]
\	Escapa el significado de los caracteres especiales

Sustitución

- Es una variante del operador "match" que permite buscar y reemplazar en la cadena original.
- Se pone 's' después de "=~ "

```
$str =~ s/buscaPatron/reemplazaPor/[modificador];
```

Ejemplo:

```
$str = "el camino es largo"
# Se cambia "es" a "no es"
$str =~ s/es/no es/
# $str es ahora: "el camino no es largo
```

Traducción de caracteres

- El operador 'tr' reemplaza los caracteres que se indiquen por otros.
- Se pone 'tr' después de "=~ "

```
$str =~ tr/caracter/reemplazaPor/;
```

Ejemplo:

```
$str =~ tr/a/b/;  # reemplaza letras 'a' por 'b'
$str =~ tr/A-Z/a-z/;  # reemplaza mayusculas por minusculas
```

Argumentos

- Para poder acceder a los argumentos de la línea de comandos que recibe un script en Perl se hace uso de un arreglo global llamado @ARGV.
- El separador de argumentos es mediante espacios en blanco (excepto cuando se pasan los argumentos con comillas dobles o simples).

```
perl script.pl arg1 "arg 2" arg3
./script.pl arg1 "arg 2" arg3
```

Lectura de entrada estándar

- Para poder leer los datos desde la entrada estándar se utiliza el archivo STDIN

- La forma en que se utiliza es:

```
1   $linea = <STDIN>;
```

Subrutinas

Son funciones que se pueden mandar a llamar cada vez que las necesitemos en lugar de repetir múltiples veces estas operaciones en el código y esto se genera con la sentencia sub.

```
1  sub funcion {
2      # Bloque de instrucciones a ejecutar
3  }
```

Para mandar a llamar cualquier subrutina se puede hacer de dos formas:

```
1  funcion();
2  &funcion;
```

Subrutinas

* NOTA: Otro uso del & al momento de llamar a una función (&funcion();) es para el uso de prototipos, los cuales permiten que las funciones se comporten como integradas. Estos prototipos permiten la reinterpretación de la lista de argumentos y el uso de & es para ignorar los efectos. Los prototipos generalmente se consideran una característica avanzada que se utiliza mejor con gran cuidado.

Si se le desean pasar argumentos a una subrutina, se deben pasar entre parentesis al momento de su llamada.

```
1  funcion("arg1", 2, @arg3);
```

Subrutinas

Para hacer uso de los argumentos dentro de la función, se hace uso de la variable especial @_.

Por defecto perl envía los argumentos por referencia, y si bien pueden no desempaquetarse, se considera buena práctica.

Manejo de archivos

- Las variables que representan a los archivos son llamados “file handles”.
- No comienzan con ningún carácter especial, son palabras simples.
- Por convención se escriben siempre en mayúsculas (ARCHIVO, FILE, FILE_OUT, SOCK).
- Viven en un espacio de nombres global, por lo que no pueden asignarse de forma local como otras variables.

File Handles

- Pueden pasarse de una rutina a otra como cadenas.
- Los manejadores estándar: STDIN, STDOUT y STDERR siempre se abren antes de que el programa comience su ejecución.
- Cualquier otro archivo que se quiera utilizar debe ser abierto primero.

File Handles

<FILE>

- El operador '<>' devuelve una línea del archivo e incluye el salto de línea ('\n')
- Cuando ya no hay más datos de entrada, el operador devuelve "undef".

- El comportamiento del operador '<>' depende de la variable global '$/' que indica el fin de línea (generalmente "\n").

```
1  # todo el archivo en una línea
2  $/ = undef;
3  $unalinea = <ARCHIVO>;
```

File Handles

- En un contexto escalar ($var) el operador de entrada lee una línea a la vez.

```
1  # lee una linea del archivo
2  $linea = <ARCHIVO>;
```

Nota: Cuando se dice que se lee una línea del archivo realmente está en un contexto de arreglo, es decir, lee todo el archivo, pero descarta todo a excepción de la primer línea.

File Handles

- En un contexto de arreglo (@arr) el operador de entrada lee todo el archivo como un arreglo, es decir, mete maneja todas las líneas como elementos de un arreglo.

```
1  # lee todo el archivo en un arreglo (cada línea como un elemento)
2  @array = <ARCHIVO>;
```

Archivos

- Para hacer uso de archivos en perl es necesario utilizar los operadores “open” y “close” para enlazar un manejador de archivo con un nombre de archive se en encuentra en el sistema de archivos.

- Abrir archivos:

```
1  open(NOMBRE, “/path/to/file”) # lectura (por defecto)
2
3  open(NOMBRE, “</path/to/file”) # lectura (explícito)
4
5  open(NOMBRE, “>/path/to/file”) # escritura (destructiva)
6
7  open(NOMBRE, ‘>>/path/to/file’) # escritura append (no destructiva)
```

Archivos

Otra forma de abrir archivos:

```
1  open NOMBRE, 'modo', '/path/to/file'
2  open(NOMBRE, 'modo', '/path/to/file')
```

Modo (carácter)	Explicación
>	Sobreescribe archivo existente (lo crea si no existe)
>>	Concatena el contenido al archivo (lo crea si no existe)
<	Lee el archivo (solo lectura)
+<	Lee y escribe (no crea y concatena al archivo)
+>	Lee y escribe (crea y sobreescribe el archivo)
+>>	Lee y escribe (crea y concatena al archivo)

Archivos

- Cerrar archivos

Siempre es necesario cerrar los archivos para evitar errores de flujo de datos.

La forma principal es:

```
1   close(ARCHIVO);
2   close ARCHIVO;
```

Manejo de errores para abrir y cerrar

- Existen diversas formas para manejarlos haciendo uso de las estructuras de control (unless), función (die, warn) o con módulos que nos permiten simplificarlo.

- Unless:

```
1  unless(open ARCHIVO, "<", "./otro.txt") {
2      die "Tampoco se puede abrir\n";
3  }
```

Manejo de errores para abrir y cerrar

- Die

```
1  open ARCHIVO, '<' , "./nuevo.txt" or die "No se puede abrir\n";
```

- Warn

```
1  close ARCHIVO or warn "Error al cerrar el archivo: $!";
```

Manejo de errores para abrir y cerrar

- Autodie

```
1  use autodie;
2  open ARCHIVO, '<' , "./nuevo.txt";
3  close ARCHIVO;
```