programa de BECAS
DGTIC
de FORMACIÓN en TIC

Seguridad Informática

Programación con Perl

Cervantes Varela Juan Manuel
Vallejo Fernández Rafael Alejandro

Referencias

- Son equivalentes a los punteros en C
- Referencian a datos contenidos en otra entidad.
- Son tratados como tipo de dato escalar.
- La definición de referencia se usa mediante el operador de referencia backslash (\)
- Tiene un mecanismo de *"papelera de reciclaje"* que consiste en un registro que posee las diversas referencias a cada elemento en memoria y las destruye cuando descubre que nadie hace referencia a él.
- Se pueden referenciar variables (creando un nuevo punto de acceso a su valor) o a valores (creando un objeto anonimo al que solo se puede acceder por la referencia).
- Para referenciar de forma anonima se deben delimitar las cadenas entre "" e iniciando con \, las listas entre [] y los hashes entre {}

Referencias

```perl
$escalar = 5;
@arreglo = qw(a b c);
%hash = ("uno", 1, 2, "dos")
$rescalar = \$escalar;  # Referencia a variable escalar
$rarreglo = \@arreglo;  # Referencia a variable array
$rhash = \%hash;        # Referencia a variable hash
```

```perl
$rescalar = \"hola";              # Referencia anonima a variable escalar
$rarray = [1,2,3];                # Referencia anonima a variable array
$rlista = {"llave1" => "dato1"};  # Referencia anonima a variable hash
```

```perl
print $rescalar, "\n";  # SCALAR(0x5618bf01a738)
print $rarreglo, "\n";  # ARRAY(0x5618bf00d9a0)
print $rhash, "\n";     # HASH(0x5618bf01a5d0)
```

Referencias

Para acceder o modificar los valores de una referencia se puede hacer mediante tres formas:

- Añadiendo el identificador como prefijo a la referencia (con ésta última entre llaves).
- Añadiendo el identificador a la referencia (quedando un doble identificador, no recomendado)
- Mediante el operador "->"

```
1  $rarray = [1,2,3,4];              # Creacion de array anonimo
2  print ${ $rarray }[2], "\n";      # 3
3  print $$rarray[2], "\n";          # 3
4  print $rarray->[2], "\n";         # 3
5  $rarray->[2] = "tres";            # o ${rarray[2]} = "tres";
6  print $rarray->[2], "\n";         # tres
7
8  @$rarray=();                  # Limpia el arreglo anonimo
```

Referencias

```
1  $val = hola;
2  print $val, "\n";    # hola
3  $rval = \$val;
4  $$rval = "mundo";
5  print $val, "\n";     # mundo
```

```
1  @array = (1, 2, 3, 4, 5);
2  $rarray = \@array;
3  print @array, "\n";      # 12345
4  $rarray->[2] = 3000;
5  print @array, "\n";      # 12300045
6  $$rarray[3] = 6;
7  print @array, "\n";      # 12300065
```

```
1  %hash = ("uno", 1, "dos", 2, "tres", 3);
2  print %hash, "\n";           # uno1do2tres3
3  $rhash = \%hash;
4  $rhash->{"dos"} = 2000;
5  print %hash, "\n";           # uno1dos2000tres3
6  $$rhash{"cuatro"} = 4;
7  print %hash, "\n";           # uno1dos2000tres3cuatro4
```

Referencias

```perl
%hash = (1, "uno", 2, "dos", 3, "tres");
print %hash, "\n";
$rhash = \%hash;
print $rhash, "\n";
# Agregar/modificar
${ $rhash }{"cuatro"} = 4;
#$$rhash{"cuatro"} = 4;
#$rhash->{"cuatro"} = 4;
# Eliminar
delete ${ $rhash }{3};
#delete $$rhash{3};
#delete $rhash->{3};
#Acceder
print ${ $rhash }{3};
#print $$rhash{3};
#print $rhash->{3};
print %hash, "\n";
```

```perl
@array = (1, 2, 3, 4);
print "@array\n";
$rarray = \@array;
print $rarray, "\n";
# Modificar
${ $rarray }[2] = 1000;
#$$rarray[2] = 1000;
#$rarray->[2] = 1000;
# Eliminar
delete ${ $array }{3};
#delete $$array{3};
#delete $array->{3};
#Acceder
print "${ $rarray }[3]\n";
#print "$$rarray[3]\n";
#print "$rarray->[3]\n";
#print "@rarray->[1,3]\n"; No funciona
#print "@$rarray[1,3]\n";
#print "@{ $rarray }[1,3]\n";
print "@array\n";
```

Referencias

```perl
@array = (
    1,
    2,
    ["a", ["b"], "c"],
    [100, 200],
    "c",
    "d"
);
print "$array[2][1][0]\n"; # $array[2]->[1]->[0]
```

```perl
%hash = ("numero", 10, "cadena", "hola");
print "$hash{\"cadena\"}\n";
@array = (1, 2, 3);
$hash{"array"} = \@array;
print $hash{"array"}, "\n";
print ${ $hash{"array"} }[0], "\n";
#print $hash{"array"}->[0], "\n";
#print $$hash{"array"}[0], "\n"; # No funciona
%new_hash = ("uno", 1, "dos", 2);
$hash{"hash"} = \%new_hash;
print $hash{"hash"}, "\n";
print $hash{"hash"}{"dos"}, "\n";
# $hash{"hash"}->{"dos"}
#print ${ $hash{"hash"} }{"dos"}, "\n";
#print $$hash{"hash"}{"dos"}, "\n"; # No funciona
```

Referencias

```perl
$hash = {"numero", 10, "dic", {1, 2, 3, 4}, "lista", [5, 6, 7]};
print $$hash{"numero"}, "\n";
print $$hash{"dic"}{1}, "\n";
#print $$hash{"lista"}->[0, 1], "\n";
print $hash, "\n";
print %{$hash}, "\n";
print ${$hash}{"lista"}, "\n";
print @{${$hash}{"lista"}}[0, 1]; # @{ ${$hash}{"lista"} }->[0, 1]
```

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.

Para retornar un valor se hace mediante la instruccion return, en caso de no retornar un valor de forma explicita, se retorna el ultimo almacenado.

```
1  sub valor {
2      $variable = 13;
3      #return 4;
4  }
5  print valor();
```

Subrutinas

Debido al "aplanado" de variables que hace perl sobre el tipo de datos, y al almacenarse los argumentos en un arreglo, no es posible enviar listas y hashes correctamente, si se quieren enviar, obligatoriamente es mediante referencia.

```
1   sub argumentos {
2       print "@_";          # 0 1 2 3 a 1
3       $var = shift @_;
4       print "$var\n";      # 0
5       $var = shift @_;
6       print "$var\n";      # 1
7       $var = shift @_;
8       print "$var\n";      # 2
9   }
10  argumentos(0, (1, 2, 3), ("a"=>1, "b">=2));
```

Subrutinas

Otro dato importante es el uso de require inserta todo el código de otro script de perl por lo que se pueden hacer un catalogo de funciones y utilizarlas en un programa.

```
1   require "./ruta_del_archivo";
```

Archivos

- Ejemplo de lectura de archivos

```perl
#!/usr/bin/perl
use warnings;

open FILE, "$ARGV[$#ARGV]" or die "no lo puedo abrir\n";

foreach (reverse <FILE>){
        print;
}

close FILE;
```

Archivos

- Ejemplo de lectura de archivos

```perl
#!/usr/bin/perl
use warnings;

$numline = 0;
open FILE, "$ARGV[$#ARGV]" or die "no lo puedo abrir\n";

while(<FILE>){
        print ++$numline.":". $_;
}

close FILE;
```

Archivos

- Escritura de archivos

Para escribir en un archivo podemos utilizar la función print donde el primer argumento es el “filehandle” y el segundo es el contenido que se escribirá.

```
1  open FIILE, ">salida.txt" or die "algo fallo\n";
2  print FIILE "prueba de escritura en archivo\n";
3  print FIILE "otra prueba\n", "dos lineas\n";
4  print FIILE "se escriben las 3 lineas?\n";
5  close FIILE;
```

Print, printf y sprintf

- **Print**

Imprime en salida estándar o en un archivo la lista de cadenas que recibe separadas por coma o concatenadas con el operador '.'

```
1  print FILEHANDLE lista de valores/variables;
2  print lista de valores/variables;
```

Print, printf y sprintf

- **Print**

Ejemplo:

```
1  print "hola:". 2334 ."\n"; # hola:2334
2  print "hola:",2334,"\n"; # hola:2334
3  print "hola:","adios"."\n"; # hola:2334
```

```
1  open (NUEVO, ">printfile.txt") or die "error";

2  printf NUEVO "hola:". 2334 ."\n"; # hola:2334
3  print NUEVO "hola:",2334,"\n"; # hola:2334
4  print NUEVO "hola:","adios"."\n"; # hola:2334
5  close NUEVO;
```

Print, printf y sprintf

- **Printf**

Maneja la forma exacta en que los números o cadenas serán formateados.

Devuelve: muestra en salida estándar o escribe en un archivo la cadena resultante (no se puede asignar a una variable).

```
1  printf FHANDLE formatoCadena, lista de valores/variables;
2  printf formatoCadena, lista de valores/variables;
```

formatoCadena:

% => indica que se va a utilizar un formateador

n1 => número/carácter con el que se llenarán los espacios

n2 => longitud total del número (incluye el punto dec)

.n3 => números a poner después del punto decimal

letra => notación que será utilizada para el formateo

Nota: no todos los compontente de formatoCadena son necesarios, depende de la notación que será utilizada y el valor que recibe.

Print, printf y sprintf

- **Printf**

Ejemplo:

```
1   printf "%.2f",15.342342533; # 15.34
```

```
1   open (FILE, ">outprint.txt") or die "error";
2   printf FILE "%.2f",15.342342533; # 15.34
3   close FILE;
```

Print, printf y sprintf

- **Sprintf**

Manipula la forma exacta en que los números o cadenas serán formateados.

Devuelve: cadena formateada (en lugar de mostrarla)

```
sprintf(formatoCadena, lista de valores/variables);
```

formatoCadena:

% => indica que se va a utilizar un formateador

n1 => número/carácter con el que se llenarán los espacios

n2 => longitud total del número (incluye el punto dec)

.n3 => números a poner después del punto decimal

letra => notación que será utilizada para el formateo

Nota: no todos los compontente de formatoCadena son necesarios, depende de la notación que será utilizada y el valor que recibe.

Print, printf y sprintf

- **Sprintf**

Ejemplo:

```
1  $str = sprintf("%06.2f", 3.1416);
2  print $str;  # $str = 003.14
```

Para realizar conversiones a hex, oct, bin:

```
1  $decimal = 24;
2  $hex = sprintf("%x", $decimal);
3  print $hex."\n";  # 18
4  $oct = sprintf("%o", $decimal);
5  print $oct."\n";  # 30
6  $bin = sprintf("%b", $decimal);
7  print $bin."\n";  # 11000
```

Funciones de conversión

- Existen funciones que permiten convertir un dato de entrada (cadena) a un número decimal para poder trabajar con ellos.
- Algunas de ellas son:
  - Cadena a decimal:

    $conv = “25”; # $conv = “25”

    $conv = “25” + 0; # => $conv = 25
  - oct($valor_octal);

    Devuelve: el valor equivalente en decimal
  - hex($valor_hexadecimal);

    Devuelve: el valor equivalente en decimal

Archivos (comprobaciones)

- Es posible realizar comprobaciones de los archivos con Perl para obtener información sobre el estado de dicho archivo.
- El valor que devuelven estas pruebas es: 1 (verdadero) o “” (falso)
- La sintaxis para realizar las pruebas es:

-**letra** “path/to/file”

-**letra** $fh

“letra” => indica la prueba que se va a realizar

Archivos (comprobaciones)

- Algunas pruebas comunes que pueden realizarse son:

```
-e "/path/to/file"  => prueba si existe el archivo
-f "path/to/file"   => prueba si es un archivo simple
-d $file            => prueba si es un directorio
-l $file            => prueba si es un enlace simbólico
-r "path/to/file"   => prueba si tiene permisos de lectura
-w "/path/to/file"  => prueba si tiene permisos de escritura
```

Archivos (comprobaciones)

- Un ejemplo de estas funciones se muestra a continuación:

```
#!/usr/bin/perl
use warnings;
$file = "entrada.txt";
unless(-e $file){
    die "El archivo \"$file\" NO existe\n";
}
print  "El archivo \"$file\" si existe\n";
```

File handle (pipe)

- En Perl es posible utilizar un manejador de archivos para abrir “pipes” de lectura y escritura. Es decir, utilizar la salida de un comando o bien, mandar la entrada desde perl a dicho comando.
- Es posible gracias a la función open.
- El primer modo es: **PIPE de lectura**

```
# Abre un pipe para leer la salida de un comando, es decir,
# ejecuta el comando $cmd y nombra la salida como FILE
open FILE, "$cmd |";
open (FILE, "$cmd |");
```

File handle (pipe)

- El segundo modo es: **PIPE de escritura**

```
# Abre un pipe para escribir como entrada a un comando, es decir,
# lo que se escriba en FILE será la entrada estándar
# STDIN del comando $cmd
open FILE, "| $cmd";
open (FILE, "| $cmd");
```

File handle (pipe)

- **Ejemplo de PIPE de lectura:**

```
1  open PIPEREAD, "ls -l |";
2  while(<PIPEREAD>){
3      # hacer lo que sea con la salida del comando: "ls -l"
4      print;
5  }
6  close PIPEREAD;
```

- Sería equivalente a ejecutar en una shell:

```
1  $ ls -l > /tmp/lsoutput.txt
```

File handle (pipe)

- Y luego con Perl:

```perl
open LS, "/tmp/lsoutput.txt";
while(<LS>){
    # hacer lo que sea con el contenido del archivo
    print;
}
close LS;
```

File handle (pipe)

- **Ejemplo de PIPE de escritura:**

```
1  open PIPEOUT, "| grep --color \"pipe\"";
2  # pone las cadenas como entrada estándar
3  # para el comando  del segundo argumento: grep --color "pipe"
4  print PIPEOUT "funcion de pipe write con perl\n";
5  print PIPEOUT "multiples lineas de entrada en pipe write";
6
7  close PIPEOUT;
```

- Sería equivalente a ejecutar en una shell:

```
1  $ echo -e "funcion de pipe write con perl\n
   multiples lineas de entrada en pipe write" | grep --color "pipe"
```

Ejecutar comandos de shell

- Es posible ejecutar comandos de Shell dentro de un script de perl.
- Para poder hacerlo existen dos formas:
- **system();**
- **operador “ ` ` ”.**

Ejecutar comandos de shell

- **Función system():**

Ejecuta el comando que está entre comillas dobles, simples o en un arreglo de cadenas.

system(“comando”);

Devuelve: el valor del código de salida de ejecución del comando, es decir, 0 si se ejecutó correctamente o cualquier otro valor para indicar un error.

Cuando se ejecuta un comando con esta función muestra también la salida en pantalla.

Ejecutar comandos de shell

- **Operador ``:**

Este operador captura la salida de un comando que esté entre los operadores ``.

```
1   $outputcmd = `comando`;
```

Devuelve: salida estándar del comando ejecutado y puede manipularse como cualquier cadena en Perl.

Ejercicios:

4. Con un script de perl obtener el archivo perteneciente a la siguiente referencia https://openphish.com/feed.txt y actualizar este archivo descargándolo cada 5 min.

5. Con el archivo anterior hacer un script en perl que verifique cuales de estos sitios están activos y guardarlos en un nuevo archivo de sitios activos.

6. Aleatoriamente obtener el código fuente de cinco de los sitios activos y buscar palabras relacionadas con phishing en la fuente especificada.

7. Generar un hash que obtenga como valor llave el dominio del sitio phishing activo y como el contenido un arreglo que contenga las urls asociadas a este.

8. Generar un reporte con estadísticas sobre los dominios involucrados en estas paginas phishing que ordene de mayor a menor las apariciones de los dominios con sitios activos de esta índole.

Scope

Es el alcance de la variable, o desde que parte del programa se puede acceder a la misma.

Al tipo de variable según su alcance, se les puede definir como variables globales o privadas (también conocidas como variables léxicas).

El uso de my declara a la variable como privada, y su scope se limita al bloque de codigo en el que se encuentre definida (delimitado por {}).

Existe our, el cual limita el alcance entre paquetes.

Scope

```perl
my $var = "hola"; # $var = "hola";
sub funcion {
    print "Subrutina a: $var\n";  # hola
    $var = "mundo";
    print "Subrutina d: $var\n";  # mundo
}
print "principal a: $var\n";      # hola
funcion();
print "principal d: $var\n";      # mundo
```

```perl
my $var = "hola"; # $var = "hola";
sub funcion {
    print "Subrutina a: $var\n";  # hola
    my $var = "mundo";
    my $new = "!";
    print "Subrutina d: $var\n";  # mundo
    print "Subrutina d: $new\n";  # !
}
print "principal a: $var\n";      # hola
funcion();
print "principal d: $var\n";      # hola
print "principal d: $new\n";      #
```

Módulos y paquetes

Use: Carga los módulos durante la compilación,

Require: Carga los modulos durante la ejecución, Usado cuando se necesita cargar grandes modulos de forma ocasional.

Paquete: Colección de código.

Módulo: Un módulo es un paquete definido en un archivo con el mismo nombre que el módulo. Cada paquete vive en su propio espacio de nombres.

Ambos están implementados para que otros módulos o programas lo puedan reutilizar.

```
1   use LWP::Simple;
```

CPAN

- Acrónimo de Comprehensive Perl Archive Network.
- Es una biblioteca que nos permite agregar módulos externos a Perl y se encuentra en meta::cpan.

- Para utilizar CPAN es necesario instalar o verificar que se tengan las dependencias necesarias:

```
1   apt update
2   apt upgrade
3   apt install build-essential curl
```

CPAN

- Para instalar un módulo con CPAN se utiliza el siguiente comando:

```
1   cpan -i nombreModulo
```

- Nota: la instalación puede tardar porque requiere instalar actualizaciones y/o módulos adicionales.

Prácticas

- Práctica 1

Elaborar un script de Perl que realice un análisis de red con nmap (nmap –sP red/24) y elabore un reporte en txt con estadísticas sobre los resultados.