

Qué es Perl ?

Perl es un lenguaje de computador interpretado muy original... a pesar de su apariencia de taquigrafía tiene mucho de los lenguajes naturales, resolver un problema en Perl es increíblemente mas fácil (y mas corto) que en cualquier otro lenguaje.

Perl no tiene esas reglas de los otros lenguajes que nos hacen perder tiempo, en Perl uno no tiene que pensar si la variable es numérica o carácter, no tiene que pensar si el valor cabe en la variable y por supuesto no hay que compilar lo que **no** significa que Perl sea lento, tampoco hay duda de que Perl se aprende rápido al menos lo básico es muy fácil hacer pequeños programas para aclarar rápidamente las dudas esta, también la

ventaja de su portabilidad, Perl existe hasta en Windows es como volver a la época en que todos los computadores tenían Basic.

Variables

Para empezar no es necesario declarar las variables previamente las variables

se pueden empezar a usar directamente en las expresiones, existen 3 tipos básicos de variables.

1. escalares

las variables escalares empiezan por \$

```
$a = 5;
```

```
$b = "xxx";
```

```
$c = $a++; # $a++ es como en C o sea
```

```
$a + 1
```

Las instrucciones terminan en punto y coma...

Perl se confunde muy fácil si se olvida el punto y coma.

Todo lo que este después de # en la misma línea es un comentario. Un escalar puede tener números, strings u otras cosas mas complicadas como referencias y descriptores.

2. arreglos

Las variables arreglos empiezan por @

```
@a = (95, 7, 'fff' );  
print $a[2];  
# imprime el tercer elemento: fff  
print @a  
# imprime: 957fff ...todo pegado
```

Sobre arreglos:

Los elementos de un arreglo son
escalares que empiezan por \$
los subíndices empiezan por 0 como en
C
el escalar \$a no tiene que ver nada con
\$a[]

Sobre print

print es una de las muchas funciones de
Perl
en Perl hay mucha flexibilidad para
escribir los argumentos

```
print ( $a, $b ); # con
```

parentesis

```
print $a, $b; # sin
```

parentesis

3. hashes o arreglos asociativos

las variables hash empiezan por %

para crear un elemento de un hash se requiere una lista de 2 valores
el primer elemento es la clave y el segundo es el valor

```
%a = ( 'x', 5, 'y', 3);  
# llena 2 elementos del hash
```

```
print ${'x'};  
# imprime: 5
```

```
print ${'y'};  
# imprime: 3
```

si la clave es un string sencillo se puede omitir las comillas

```
 ${x} es lo mismo que ${'x'}
```

si se reasigna un elemento se pierde el valor viejo

```
%a = ('x', 5, 'y', 3 );  
${x}=7;  
print ${x};  
# imprime: 7
```

Sobre los hashes

los elementos se accesan por claves
no hay claves duplicadas

Valores

Los valores son las cosas que uno mete a las variables, en Perl los valores se guardan separados de las variables e incluso cada valor tiene un contador que indica cuantas variables lo están usando cuando el contador es cero el valor desaparece.

Valores interesantes:

1. los **strings**

los strings pueden escribirse con comillas dobles o simples

```
$a = "abcd";
```

```
$a = 'abcd';
```

cuando se usa doble comilla se pueden intercalar variables escalares y arreglos en el resultado final la variable se substituye por su valor.

```
$a = 'pueblo'
```

```
print "hola $a";
```

```
# imprime: hola pueblo
```

```
print 'hola $a';
```

```
# no hay substitucion en comillas
```

simples...

```
# imprime: hola $a
```

uno se puede preguntar como se substituye una variable entre

letras sin que se altere el nombre de la variable p.e. como se coloca \$a antes de una letra como "s" .

```
"abc$as";
```

```
# no sirve... trata de substituir la variable
```

\$as

```
"abc${a}s"
```

```
# substituye correctamente $a
```

entre las comillas dobles se pueden tambien poner
caracteres especiales

como \n (salto de linea), \t (tabulador), \b (backspace),

```
print "aaa \n";
```

```
# salta linea
```

```
print 'aaa \n';
```

```
# no salta... imprime: aaa \n
```

entre las comillas doble. se tiene que **escapar** los
caracteres

que tienen significado especial en Perl... como \$, ", \

```
$a = 1234;
```

```
print "valor=$a\$";
```

```
# imprime: valor=1234$
```

la "escapada" se hace con backslash antes del
caracter

entre las comillas simples solo se tiene que escapar

'y \

```
print 'abc\s';
```

```
# imprime: abc's
```

cuando se substituye un arreglo, Perl regala un
separador que es

un blanco

```
@a = (95, 7, "fff" );
```

```
print "val=@a" ;
```

```
# imprime: val=95 7 fff
```

el separador es realmente el valor de la variable
escalar "\$" ...

"\$" puede ser reasignada

```
"$" = ',';  
print "val=@a";  
# imprime: val=95,7,fff
```

2. las listas

p.e. (2, 7, 'sss')

```
$a = (2, 7, 'sss');  
# $a queda con el último valor: 'sss'
```

```
@a = (2, 7, 'sss');  
$a = @a;  
# $a queda con el número de elementos
```

de @a: 3

note que (2, 7, 'sss') es un valor
mientras que @a es un arreglo - recuerde que las
variables y los

valores son 2 cosas distintas

existen abreviaturas para algunas listas

```
$a = (2..7);  
# $a queda con (2,3,4,5,6,7);
```

```
$a = ('a'..'e');  
# $a queda con ('a','b','c','d','e')
```

3. strings con comillas invertidas

las comillas invertidas se comportan como en los
shelles de Unix

```
$a = `ls -l`;
```

```
# $a queda con la salida del comando "ls
```

```
-l"
```

entre las comillas invertidas se pueden intercalar
variables

escalares entre las comillas dobles se puede
intercalar

comillas invertidas

asociar un valor con una variable se llama **bind...**

```
$a = 5;
```

```
# aqui 5 es el valor de $a
```

```
$b = $a;
```

```
# aqui el valor 5 de $b es otro valor
```

para que se utilice un solo valor para 2 variables se
necesita

utilizar referencias que se ven mas adelante.

Más sobre arreglos

Los subíndices positivos recorren los elementos al derecho

```
@a = ('a'..'e');
```

```
# $a[0] es 'a' $a[4] es 'e'
```

Los subíndices negativos recorren los elementos al revés

```
@a = ('a'..'e');
```

```
# $a[-1] es 'e' $a[-5] es 'a'
```

un arreglo en **contexto** escalar retorna el número de elementos

```
@a = ('a'..'e');  
$n = @a;  
# aquí $n es 5
```

Lo del contexto es una característica de Perl.
Perl evalúa una expresión según el uso que se piensa dar a la expresión: contexto escalar o contexto de

lista

subarreglos:

```
@a = ('a'..'e');  
@b = @a[3, 0];  
# @b = ('d', 'a');
```

```
@c = @a[2-5];  
# @a[2,3,4,5] o @a[2..5]
```

es posible colocar una lista de variables escalares a la izquierda del igual

```
($a, $b) = @x  
# $a queda con el valor de $x[0]  
# $b queda con el valor de $x[1]
```

es posible colocar un subarreglo a la izquierda del igual

```
@a[3, 0] = @a[0, 3];  
# intercambia los elementos 0 y 3
```

la función **join** convierte un arreglo en un escalar

```
@a = ('a'..'e');  
$a = join ":", @a  
# $a queda con "a:b:c:d:e";
```

la función **split** convierte un escalar en un arreglo... muy útil para

separar los campos en un registro de un archivo texto

```
$a = 'a:b:c:d:e';  
# no le recuerda esto al /etc/passwd?
```

```
@a = split /:/, $a  
# @a queda con ('a', 'b', 'c', 'd', 'e')
```

aqui el segundo parámetro es mas que un string...
es una regex... regex es un cuento largo que se ve

mas adelante

la función **shift** entrega el 1er elemento del arreglo y ademas le
quita ese elemento al arreglo

```
@a = ('a'..'e');  
$b = shift @a;  
# $b queda con 'a'  
# @a queda con ('b'..'e');
```

la función **pop** entrega el último y bota el último del arreglo

```
@a = ('a'..'e');  
$b = pop @a;  
# $b queda con 'e'  
# @a queda con ('a'..'d');
```

se puede intuir que hacen las funciones **unshift** y **push...**

```
unshift @a, 'a';  
# agrega 'a' al principio del arreglo
```

```
push @a, 'e';  
# agrega 'e' al final del arreglo
```

unshift y push pueden tambien agregar una lista de

elementos

en lugar se un solo elemento

la función **splice** permite extraer un subarreglo
y modificar a la vez el arreglo original...

```
@a = ( 'a'..'e');  
@b = splice ( @a, 1, 2);  
# @b queda con 2 elementos de @a: $a[1] y $a[2];  
#   ( 'b', 'c')  
# @a queda sin esos 2 elementos:  
#   ( 'a', 'd', 'e' );
```

splice con un argumento mas sirve para parchar el
arreglo original

```
@a = ( 'a'..'e');  
@b = ( 1..3);  
splice ( @a, 2, 1, @b);  
# @a queda con ('a', 'b', 1, 2, 3, 'd', 'e');  
# se cambio 'c' por (1, 2, 3)
```

tambien se puede parchar sin botar nada...

```
@a = ( 'a'..'e');  
@b = ( 1..3);  
splice ( @a, 2, 0, @b);  
# @a queda con ('a', 'b', 1,2,3, c', 'd', 'e');
```

Verdadero y falso

Como en C, cualquier expresión tiene un significado lógico,
p.e. las asignaciones tienen el valor de lo asignado.

los valores falsos son:

1. los strings "" y "0"

2. el número 0.

3. el valor "no definido" de una variable...
cuando existe la variable pero no tiene valor

obviamente todo lo demás valores son verdaderos

la funciones **defined** se usa para averiguar si una variable esta
definida

```
$a = 5;  
print "a definida" if (defined $a);  
print "b no definida" if (!defined $b);
```

nota sobre ese if:

el if escrito al revez parece muy cómodo porque uno

se

ahorra las llaves

en el if normal nunca se pueden omitir las llaves

```
if ( defined $b )  
{  
    print "b definido";  
}
```

mas adelante se muestra otra forma
de hacer lo mismo que me gusta mas...

```
defined $b  
and  
print "b definido";
```

Ojo que la función **undef** no es lo contrario de defined... lo que
hace

realmente es volver el argumento "no definido"... elimina el **bind**

entre la variable y el valor... el valor desaparece si su contador de usuarios queda en cero

Expresiones lógicas

una expresión lógica es una expresión cuyo valor es verdadero o falso usualmente las expresiones lógicas se usan en condiciones

```
if ( $a and $b )
{
    print "A y B son verdaderos";
}
```

```
if ( $a or $b )
{
    print "A o B son verdaderos";
}
```

existen también los operadores **&&** y **||**
como en C... que tiene mas prioridad que and y or...

las expresiones lógicas tiene otro uso muy interesante usándolas como una instrucción en si misma...

```
$a and $b;
# no no... no es un error !
```

si \$a es falso toda la expresión anterior es falsa por lo tanto no se evalúa el elemento \$b.

si \$a es verdadero se tiene que evaluar \$b para conocer el valor de la expresión y eso aunque el valor de la expresión total no se utiliza para nada...

```
$a
and
print "A es verdadero";
```

el print solo se hace si \$a es verdadero
equivale a

```
print "A es verdadero" if $a;
```

el otro caso es \$a or \$b:

```
$a  
or  
print "A es falso";
```

el print solo se hace si \$a es falso
equivale a

```
print "A es falso" if !$a;
```

o tambien a

```
print "A es falso" unless $a;
```

Operadores

Operadores lógicos

operadores para comparar números... como en C

```
$a == $b and print "A igual a B";  
$a != $b and print "A distinto de B";  
$a >= $b and print "A >= B";
```

para comparar strings se usan otros operadores

```
$a eq $b and print "A igual a B";  
$a ne $b and print "A distinto de B";  
$a ge $b and print "A >= B";
```

el siguiente ejemplo muestra porque se necesita

distinguir

una comparación numérica de una comparación de

strings

```
$a=5; $b=49;  
$x = ($a gt $b)  
# $x queda 1 ( verdadero)
```

```
$x = ($a > $b)  
# $x queda "" ( falso)
```

comparación de números (spaceship operator)

```
$x = $a <=> $b  
# $x queda con -1 si $a < $b  
# $x queda con 0 si $a == $b  
# $x queda con 1 si $a > $b
```

comparación de strings

```
$x = $a cmp $b  
# $x queda con -1 si $a lt $b  
# $x queda con 0 si $a eq $b  
# $x queda con 1 si $a gt $b
```

operador ternario

una abreviatura de las 3 partes de algunos if la
condición , la acción para verdadero y la acción
para falso separados por ? y :

```
@a > 5 ? print "a es > 5": print "a no es  
> 5";
```

```
o
```

```
$txt = $a > 5 ? "a es > 5" : "a no es > 5";  
print $txt;
```

operadores de strings

repetir strings con operador "x"

```
$a = "y";  
$b = $a x 5  
# $b queda con "yyyyy";
```

concatenar strings con operador "."

```
$a = "abc"  
$b = $a . "def" ;  
# $b queda con "abcdef"
```

operador de rango ".." para hacer listas

```
@a = "ay" .. "bb";  
# @a queda con ("ay", "az", "ba", "bb")
```

```
@a = 1..5;  
# @a queda con (1, 2, 3, 4, 5)
```

ojo no mezcle letras y números

ojo no mezcle mayúsculas y minúsculas

Funciones

En Perl se puede definir una función es cualquier parte la función solo se ejecuta cuando se llama expresamente la función se define con sub

```
sub fun1  
{  
    $a = shift;  
    # shift asume el arreglo @_  
    # @_ es el arreglo que tiene los  
  
    # que se dan al llamar la función  
  
    $y = 2 * $a;
```

argumentos

```
return ( $y );  
# devuelve ese valor al que llamó la  
función  
}
```

la función se llama con &

```
$x = 5;  
$z = &fun1( $x );  
# pasa $x como único elemento de @_  
# osea que $z queda con 10
```

una función que no tiene "return" de todas maneras
retorna algo retorna el valor de la última expresión esto quiere decir
que la función fun1 arriba no necesita el "return" no es obligación que
el llamador utilice el valor del return como en C, los parámetros de una
función se pasan por referencia ósea que si se modifica \$_[1] se
alteraría el 2do parámetro usado en la expresión llamadora, algo
peligroso

```
sub fun1  
{  
    $_[0]=7;  
    # altera el 1er parámetro en el llamador  
}
```

```
$a = 5;  
&fun1($a);  
print $a;  
# imprime: 7
```

Bloques

Un bloque son expresiones dentro de llaves

las funciones que vimos son bloques pero también puede haber bloques

sin la palabra sub una de las razones para tener bloques es tener variables locales

que desaparecen cuando el bloque termina.

```
$a = 5;
# variable global que nunca muere

{
    $b = 7;
    # variable global que nunca muere

    my ( $c ) = 3;
    # "my" crea una variable local
    # que solo existe en este bloque

    &fun1 ()
    # $c no es visible dentro de fun1
}
print $a; # imprime: 5
print $b; # imprime: 7
print $c; # error... variable no existe

sub fun1
{
    print $a; # imprime: 5
    print $b; # imprime: 7
    print $c; # error... variable no existe
}
```

La función **my** es la mas utilizada para definir variables locales...
las variables "my" o lexicas son visibles solo dentro del bloque...

no son visibles fuera del bloque... tampoco son visibles a las funciones que se llaman desde el bloque...

La función **local** se usa para definir otras variables locales estas variables tapan provisionalmente las variables globales estas variables **si** son visibles a las funciones que se llamen desde el bloque.

```
$a = 5; # variable global que nunca muere
{
    local ( $a ) = 3;
    # el viejo valor 5 se guarda
    # para reponerlo cuando este bloque
    # termine

    local ( $b ) = 7;
    # como $b no existia entonces
    # se guarda "no definido"
    # para recordarlo cuando este bloque
    # termine

    &fun1 ();
    # en fun1 se puede usar $a y $b
}
print $a; # imprime: 5
print $b; # error... variable "no definida"

sub fun1
{
    print $a; # imprime: 3
    print $b; # imprime: 7
}
```

```
}
```

Más funciones integradas

funciones de strings

chop (\$a);

bota el último carácter del valor de \$a

```
$a = "abcdef";
```

```
chop ( $a ) ; # $a queda con
```

```
"abcde";
```

muy usado para quitar el carácter "\n"

al final de una línea de un archivo texto

length (\$a)

devuelve la longitud del valor de \$a

```
$a = "abcdef";
```

```
print length ( $a ); #
```

```
imprime: 5
```

index (\$a , \$x);

devuelve la posición de \$x en \$a

```
$a = "abcdef";
```

```
$b = index ( $a, "cd" );
```

```
print $b; # imprime: 2
```

uc (\$a);

devuelve los caracteres de \$a en

mayúsculas

lc (\$a);

devuelve los caracteres de \$a en

minúsculas

substr (\$a, \$pos, \$len);

empezar

extrae un string de otro
el 2do parámetro es la posición para

el 3er parámetro es la longitud

```
$a = "abcdef";  
print substr ($a, 2, 3);  
# imprime: cde
```

asignación

interesante uso de substr al lado izquierdo de una

```
$a = "abcdef";  
substr ( $a, 2, 3 ) = "xy";  
# cambia "cde" por "xy"
```

```
print $a  
# imprime: abxyf
```

funciones de arreglos

@b = map expresion, arreglo

devuelve un arreglo después de aplicar la expresion
a cada elemento del arreglo en la
expresion "cada elemento" es \$ _

```
@a = ('ax'..'bc');  
@b = map uc($_), @a ;  
print "@b";  
# imprime: AX AY AZ BA BB
```

BC

@b = grep expresion, arreglo

devuelve un subarreglo de @a que

contiene los elementos

donde la expresión es verdadera.

```
"c1", "c2");
```

```
@a = ("a1", "a2", "b1", "b2",
```

```
@b = grep /^b/, @a;
```

```
print "@b";
```

```
# imprime: b1 b2 ... los que
```

empiezan por 'b'

el arreglo que devuelve grep se puede también usar como un valor lógico falso si ningún elemento del arreglo cumple la condición.

```
@b = sort ( @a )
```

devuelve un arreglo ordenado

ojo que el ordenado es @b... @a sigue

desordenado...

claro que se puede escribir

impunemente: @a = sort @a

```
@b = reverse ( @a )
```

devuelve un arreglo invertido

```
@b = reverse sort @a
```

```
# @b queda con @a ordenado
```

descendente

funciones de hash

```
@b = each ( %a );
```

@b es una lista de 2 elementos (\$key,

\$value) correspondiente

a un elemento del hash %a cuando se

llama varias veces con

el mismo hash itera sobre todos los

elementos del hash.

```
while ( ($k, $v ) = each ( %a ) )
{
    print ( "key=$k val=$v \n");
}
```

... aunque yo prefiero hacer lo anterior

asi:

```
foreach ( keys %a )
{
    print ( "key=$_ val=${a[$_]}
\n");
}
```

foreach trabaja sobre un arreglo...

recorre el bloque por

cada elemento del arreglo... \$_ es cada

elemento del arreglo...

tambien es posible usar foreach con otra

variable distinta de \$_

```
foreach $k ( keys %a )
{
    print ( "key=$k val=${a{$k}}
\n");
}
```

Enunciados

Un "if" mas completo observe el **elsif** el **elsif** ahorra paréntesis y ahorra indentaciones

```
if ( /^[1..3]/ )
{
    print "$_ es cta de balance\n";
}
```

```

elseif ( /^[4..7]/ )
{
    print "$_ es cta de resultado\n";
}
else
{
    print "$_ es cta de orden\n";
}
# las condiciones arriba son "expresiones
regulares"...

# uno de los temas mas fascinantes de Perl... que
veremos

# mas adelante

```

nuevamente el foreach...

```

@a = (1..5);
foreach ( @a )
{
    $_ == 3
    and
    $_ = "x";
}
print "@a";
# imprime: 1 2 x 4 5

```

ojo que \$_ es un alias temporal de cada elemento del arreglo osea que si se modifica \$_ se altera el arreglo pero también \$_ es variable tipo "my" que no afecta ningún \$_ global... ni es visible en subrutinas que se llamen desde dentro del foreach.

Palabras de salto

last: salta fuera del bloque actual

next: omite las instrucciones que siguen en el bloque

y comienza una nueva iteración
redo: reinicia el bloque

un while mas complicado...observe el bloque "continúe"

```
while ( /^d/ )  
{  
    /^47/ and next;  
  
    print "empieza por digito\n";  
} continúe  
{  
    # el código entre estos corchetes se hace  
  
    # de repetir la condición del while...  
    # aun si se ejecuta el next de arriba  
  
    <f1>;  
    # esto lea un registro de un archivo f1  
}
```

antes

Referencias

Las referencias son escalares que **apuntan** al valor de otra variable "apuntan a" significa "tiene la dirección de"...

```
$ra = $a; # referencia a escalar  
$rb = \@b; # referencia a arreglo  
$rc = \%c; # referencia a hash  
$rx = $rb; # referencia a referencia
```

también hay referencias a función y referencias a objetos...

las referencias interesantes son a arreglos y a hashes...

veamos otra forma de crear una referencias a arreglo...

observe el paréntesis cuadrado

```
$rb = [ 'e1', 'e2', 'e3'];
```

aquí el array no tiene nombre...

\$rb es una referencia a un **arreglo anónimo**...

otra forma de crear una referencia a hash... observe las llaves

```
$rc = { k1 => 'v1', k2 => 'v2' };
```

aquí el hash no tiene nombre...

\$rc es una referencia a un **hash anónimo**...

cuando una referencia es dereferenciada se obtiene el dato real

```
$ra = $a; # referencia a escalar
```

```
$rb = @$b; # referencia a arreglo
```

```
$rc = %c; # referencia a hash
```

```
$rx = $rb; # referencia a referencia
```

`#{ $ra }` es la desreferencia de \$ra... el valor de

\$a

`@{ $rb }` es la desreferencia de \$rb... el valor de @a

`@{ $ra }` es un error porque \$ra apunta a un

escalar

`%{ $rc }` es la desreferencia de \$rc... el valor de %c

veamos una manera de acceder los elementos de un hash usando una referencia muy útil si el hash es anónimo.

```
$rc = { a => 1, b => 2 };
```

```
# $rc es una referencias a un hash anónimo
```

```
print $rc->{a};
```

```
# imprime: 1
```

entendió bien si ud es nuevo en referencias asegúrese que

lo anterior... se lo digo por experiencia

la función **ref**

devuelve un string que indica el tipo del referenciado

```
$ra = \ $a; # referencia a escalar
```

```
$rb = \@b; # referencia a arreglo
```

```
$rc = \%c; # referencia a hash
```

```
$rx = \ $rb; # referencia a referencia
```

```
$rf = \&f; # referencia a función
```

```
ref ( $ra ); # devuelve "SCALAR"
```

```
ref ( $rb ); # devuelve "ARRAY"
```

```
ref ( $rc ); # devuelve "HASH"
```

```
ref ( $rx ); # devuelve "REF"
```

```
ref ( $rf ); # devuelve "CODE"
```

devuelve falso si el operando de ref no es una referencia, ref

```
"no definido"
```

la función **bless**

cambia el tipo de una referencia a otro tipo...

muy utilizado para crear **clases**

```
$rc = { ano => 1995, marca => 'renault', puertas => 4  
};
```

```
ref ( $rc );
```

```
# devuelve "HASH"
```

```
bless $rc "CARRO";
```

```
ref ( $rc );
```

```
# devuelve "CARRO"
```

esto tiene mas sentido cuando se hable de objetos
y paquetes.

diferencia entre trabajar con variables y trabajar con referencias...

```
$a = 5;  
$b = $a;  
# $b recibe una copia del valor de $a  
$a = 7;  
# cambio el valor de $a  
print "$a $b\n";  
# imprime: 7
```

5

```
# un cambio en $a no afecta a $b  
$a = 5;  
$ra = $a;  
$rb = $ra;  
$a = 7;  
# cambio el valor de $a  
print "$a $ra $rb \n" ;  
# imprime 7 7 7  
# las referencias si comparten en mismo valor  
# el valor 7 aqui tiene 3 usuarios
```

las referencias a arreglos son muy útiles cuando se guardan en
un arreglo porque ese arreglo simula un arreglo de n

dimensiones...

```
@x1 = ( 5, 6, 7 );  
@x2 = ( 2, 3, 4 );  
@x = ( \@x1, \@x2 );
```

en este caso el arreglo @x quedan como de 2

dimensiones

```
$x[0]->[0] es 5
```

```
$x[1]->[2] es 4
```

programadores y como Perl lo hicieron programadores para

```
$x[0]->[0] se puede escribir $x[0][0]
```

```
$x[1]->[2] se puede escribir $x[1][2]
```

Archivos

Abrir archivos

```
open f1, "auxytd.98.3";
```

el abre un archivo de lectura... el archivo se maneja con

descriptor f1 ... f1 es como el descriptor de c...

otras formas de open

```
open f1, "< auxytd.98.3";
```

```
# abrir para leer...
```

```
# el < se puede omitir...
```

```
open f1, "> auxytd.98.3";
```

```
# abrir para escribir
```

```
open f1, ">> auxytd.98.3";
```

```
# abrir para agregar
```

leer un archivo texto... archivos donde cada línea termina en "\n"
el operador diamante "<>" lee una línea del archivo

```
open f1, "archivo.98";
```

```
while ( <f1> )
```

```
{
```

```
    print;
```

```
}
```

aquí <f1> llena \$_ con una línea del archivo...

que posteriormente se imprime con print...

en Perl casi todo el mundo asume \$ _ en realidad <f1> en **contexto escalar** leen una línea y en **contexto lista** lee todo el archivo.

```
open f1, "auxytd.98.3";  
@a = <f1>;  
# @a tiene todo el archivo  
# cada elemento de @a es una línea del
```

archivo

escribir en un archivo con **print**

```
print descriptor @lista  
ojo que no hay coma entre descriptor y @lista  
si no hay descriptor se asume STDOUT  
si no hay @lista se asume $ _
```

Regex

Regex es una abreviatura de expresión regular una expresión regular es una forma general de describir un patrón de caracteres que queremos buscar en un string usualmente el patrón se escribe entre salsees las regex se utilizan en 2 clases de expresiones:

1. match del patrón en un string...

aquí la regex es una expresión lógica...
devuelve verdad si el string contiene un

patrón

```
$a = "abcdef";  
$a =~ /bc/; # es verdadero  
$a =~ /ba/; # es falso
```

"=~" se llama el operador de **bind**...

"!~" es la negación de la

expresión

```
$a = "abcdef";  
$a !~ /bc/; # es falso
```

```
$a !~ /ba/; # es verdadero
```

2. substitución... s//

muy útil para remendar

strings...

```
$a = "<option>mod";  
$a =~ s/>/ selected>/  
print $a;  
# imprime: <option selected>mod
```

cuando el escalar es \$_ se omite \$_ y =~...

```
$_ = "abcdef";  
/bc/;      # es verdadero  
s/cd//;  
print; # imprime: abef
```

cuantificadores

se usan para indicar que algunas letras se repiten

* : cero o mas del anterior caracter

+ : uno o mas del anterior caracter

? : cero o un del anterior caracter

{3,5} : mínimo 3 y máximo 5 del caracter anterior

{3,} : mínimo 3 del caracter anterior

{,5} : máximo 5 del caracter anterior

```
$_ = "abcccd";  
/c+d/ ; # es verdadero
```

puntos de referencia

^ : es el comienzo del string

\$: es el final del string

\b : es un borde de una palabra [palabras son letras

números y _]

```
$_ = "abcdef";  
/^abc/; # es verdadero
```

clases de caracteres comunes

línea

- . : un caracter cualquiera
- \s : un espacio en blanco , tabulador o salto de línea
- \S : un caracter no blanco, no tabulador y no salto de línea
- \d : un digito
- \D : un no digito
- \w : un caracter de palabra: digito letra o _
- \W : un caracter que no es de palabra

```
$_ = "d15";  
/\d+$/ ; # es verdadero
```

clases de caracteres a la medida

negación

- [abcef] : uno de esas 5 letras
- [a-f] : lo mismo que el anterior
- [0-9] : es lo mismo que \d
- [\t \n] : es lo mismo que \s
- [a-zA-Z_] : es lo mismo que \w
- [^a-zA-Z_]: es lo mismo que \W ... aquí ^ significa

```
@a = ( 1..10);  
foreach ( @a )  
{  
    /^1-3/  
    and  
    print "$_:";  
}
```

imprime: 1:2:3:10:

caracteres especiales

\. : punto
\ : backslash
\n : salto de línea
\t : tabulador
\\$: signo pesos

Más sobre regex

memoria de matchs...

los paréntesis se usan para almacenar los matchs
en las variables \$1, \$2, \$3, hasta \$9...

```
$_ = "1995 renault azul";  
s/^(w+)s+(w+)/$2 $1/; # intercambia
```

1995 y renault

```
print $_;          # imprime: renault
```

1995 azul

```
print $1;          # imprime: 1995
```

también es posible sacar los matchs a un arreglo

```
$_ = "1995 renault azul";  
@a = /^(w+)s+(w+)/;  
print "@a"; # imprime: 1995 renault
```

como si ya no fuera bastante, las regex tienen también **opciones...**

/g : indica que haga varios "match"

```
$_ = "f1=abc test=on";  
s/= / ;  
# $_ queda con "f1 abc test=on"  
$_ = "f1=abc test=on";  
s/= /g ;
```

```
# $_ queda con "f1 abc test on"
$_ = "1995 renault azul";
@a = /^(w+)/g;
# @a queda con 3 elementos
claro que el split me parece mejor...
@a = split;
# la regex default de split es / +/
```

/i : ignore mayúsculas y minúsculas

```
$_ = "Francisco francisco";
s/francisco/pacho/ig;
# $_ queda con "pacho pacho"
```

s///e : ejecuta la segunda expresión y su valor lo

utiliza

para remplazar el patron...

```
$_ = "largo= 15";
s/(d+)/$1 * 4/e;
print;
# imprime: largo= 60
```

el operador **tr** se usa para traducir caracteres...

tiene un parecido con la substitución en regex

```
$a = "f1=abc test=on";
tr/= / / ;
# $a queda "f1 abc test on"
%x = split / /, $a;
# ${f1} queda con "abc"
# ${test} queda con "on";
```

Variables especiales

Perl tiene toda su maquinaria a la vista en unas variables especiales variables que afectan arreglos

`$[` es el subscrito base de los arreglos
[default es 0]

`$"` el separador de elementos cuando se substituye un arreglo en un string de comilla doble
[default es espacio]

variables utilizadas en archivos

`$.` contiene el último número de línea leído
`$/` terminación de registro de entrada [default es '\n']

`$|` si es diferente de cero se vacía el buffer de salida después de print o write (default es 0)
se debe colocar en 1 para hablar con un puerto de tcp/ip...

variables usadas con patrones

`$&` contiene el último string que hizo match
`$+` contiene el string que coincidió con el último paréntesis que hizo match
`$1, $2, $3...` memoria de lo matches de los paréntesis

variables usadas en impresión

`$\` se agrega al final del print (default nulo)

variables relacionadas con procesos

	\$0	el nombre del script de Perl
error	\$!	número de error o string con el texto del error
del programa	%ENV	hash que tiene las variables de ambiente
cgi-bin...		p.e. \$ENV { QUERY_STRING) que lo llena el apache antes de llamar un
URL del Netscape		tiene lo que va después de la ? en el

variables diversas

	\$_	parámetro default de muchas funciones es como el "que" del español...
comando	@ARGV	argumentos de la línea de comando... es como dar split / +/ a la línea de

Lo bueno...

Paquetes

Un paquete es un espacio de nombres los espacios de nombres permiten que nosotros utilicemos código de otras personas sin que las variables de nosotros se confundan con las variables de la otra persona

```

package C110;
# estamos en el espacio de nombres C110

$a = 5;          # variable del paquete

C110
fun1             # función del paquete C110
{
    print "$a\n";
}

package D110;
# ahora estamos en el espacio de nombres D110
# ...salimos del paquete C110
$a = 7;         # esta $a es del paquete D110
print $a;       # imprime 7
print $C110::a;
# imprime 5
# note como accesamos el espacio de nombres

C110...
# note el $ y los ::
C110::fun1;     # llama a fun1 de C110...imprime: 5
fun1 C110;      # llama a fun1 de C110...imprime: 5
C110->fun1;     # llama a fun1 de

C110...imprime: 5

```

observe las 3 formas de llamar la función...

cuando no usamos "package" estamos trabajando en el espacio de nombres main" como un paquete generalmente se hace para ser reutilizado muchas veces se guarda en un archivo libreria de extension .pl como p.e. cgilib.pl y los programas que lo quieren usar lo invocan con **requiere**

```
requiere "cgilib.pl";
```

la función "requiere" lee el archivo "cgilib.pl" si este no ha

"package" pero

sido leído antes... el archivo no tiene que tener

que termine

si debe devolver verdadero... osea que lo mejor es

```
con: return 1
```

moda de los

las librerías ya no se usan tanto... porque llegó la

objetos que en Perl se implementan con módulos...

ojo... parámetro adicional que reciben las funciones de un paquete.

```
package C110;  
sub fun2  
{  
    print "fun2 recibio @_ \n";  
}
```

```
package D110;  
C110::fun2("xyz");  
# llama a fun2...imprime: fun2 recibio xyz
```

funciones de

esta forma de llamar funciones de paquete no se utiliza usualmente porque, como veremos enseguida, las

adicional...

paquete se escriben para recibir un parámetro

```
C110->fun2("xyz");  
# llama a fun2...imprime: fun2 recibio C110 xyz
```

```
fun2 C110("xyz");  
# equivalente al anterior... C110->fun2 ("xyz")
```

observe que cuando se llama con C110->fun2, fun2
recibe un

parámetro adicional...el nombre del paquete... "C110"mas
adelante veremos otra forma mas de llamar fun2 \$r->fun2()
donde \$r es una referencia a un objeto C110 en este caso el
parámetro adicional que recibe fun2 es la
referencia \$r...

Un **módulo** es un paquete en un archivo de su mismo
nombre y extensión .pm... los nombres de los módulos empiezan por
mayuscula.

p.e. el módulo CARRO debe estar en el archivo
CARRO.pm

para utilizar un módulo en un programa se utiliza **use** la
función "use" es como un "requiere" pero que ademas ejecuta una
función del módulo llamada **import**... que vamos a ignorar por ahora
en seguida veremos como los módulos se utilizan pra representar
objetos.

Objetos

Una clase es una abstracción de un objeto por ejemplo la
clase CARRO, una clase esta compuesta de propiedades y métodos
que para nosotros corresponden muy bien a variables y funciones es
bueno distinguir la clase CARRO y un objeto de la clase CARRO la
clase carro podría tener la propiedad velocidad y la función acelerar
para aumentar la velocidad un carro específico podría
tener velocidad=50 y llamar la función acelerar(7) para que cambie a
velocidad=57; los módulos se utilizan para representar clases porque

los paquetes aíslan muy bien las variables y las funciones... tal como se desea con la moda de los objetos...lo que se llama "encapsulamiento" el ejemplo que sigue utiliza 2 archivos:

1. archivo CARRO.pm... para representar la clase CARRO

```
package CARRO;
# la clase CARRO tiene 3 métodos...
# o el módulo CARRO tiene 3 funciones:
# 1. new: crea objetos tipo CARRO
# 2. acelerar : cambia la velocidad de un objeto
CARRO
# 3. status : muestra la velocidad de un objeto
CARRO
# el módulo CARRO tiene una variable
# 1. vel: la velocidad del carro

sub new
# "new" es el nombre preferido para crear
# instancias de una clase... un recuerdo de c++
{
    my ( $rc );
    $rc = { vel => 0 };
    # un hash es lo mas recomendado para
representar
    # un objeco... recuerde que las llaves
crean una
    # referencia a un hash anónimo

    bless $rc , "CARRO"
    # recuerde bless de referencias
    # $rc ahora es una referencia a un objeto
tipo
    # "CARRO"
    return $rc;
}
```

```

# devuelve una referencia tipo
"CARRO"...

# después de esto $rc muere... pero el hash
# anónimo no muere si el llamador utiliza
el

# valor del return...
}
# si la función new es llamada de nuevo devuelve
# también una referencia "CARRO"... pero a un carro
# distinto

sub acelerar
{
    my ( $rc ) = shift;
    # el 1er shift entrega el parámetro
adicional que
    # Perl antepone a los parámetros que
uso llamador...
    # en este caso la referencia con que
nosotros llamamos
    # la función en el 2do archivo...

    $rc->{vel} += shift;
    # el 2do shift entrega el parámetro
    # con que nosotros llamamos la función
en el 2do
    # archivo
}
sub status
{
    my ( $rc ) = shift;
    print "velocidad=$rc->{vel}\n";
}

```

```
return 1;  
# para que no falle el "use"...
```

2. archivo pd0010.pl... que utiliza el módulo CARRO

```
#!/usr/bin/perl -w  
# ojo la anterior linea no es comentario  
# el -w es un parámetro de Perl... para que de buenos  
# warnings
```

```
use CARRO;  
# como no usamos package... estamos en el package
```

"main"

```
$x = new CARRO;  
# lo mismo que CARRO->new();  
# $x queda con una referencia a un tipo "CARRO"...
```

```
$y = new CARRO;  
# $y es otro CARRO distinto
```

```
$x->status;  
# otra forma de llamar una función del paquete
```

"CARRO"...

```
# ...con una referencia tipo "CARRO"..  
# imprime: velocidad=0
```

```
$x->acelerar(50);  
# el primer parámetro que "acelerar" recibe es $x  
# el segundo parámetro que "acelerar" recibe es 50
```

```
$x->status;  
# imprime: velocidad=50
```

```
$y->status;  
# imprime: velocidad=0
```

En este ejemplo vemos como el módulo CARRO representa la clase CARRO y usando new de la clase CARRO creamos 2 objetos tipo CARRO: \$x y \$y podemos complicar un poco la función new de CARRO.pm para que reciba parámetros pero hay que anotar que Perl le pone un parámetro adicional en este caso el string "CARRO", que por ahora no lo necesitamos.

```
sub new  
{  
    my ( $class ) = shift;  
    # si new se llama como CARRO->new,  
$class queda  
    # con "CARRO"... aparentemente esto  
no es  
    # ninguna información... pero se vuelve  
muy útil  
    # cuando se trabaja con objetos  
"heredados" de  
    # CARRO... herencia es otro de los  
conceptos  
    # modernos de objetos  
  
    my ( $vel, $marca ) = @_  
    my $rc = { vel => $vel, marca =>  
$marca };  
  
    bless $rc, $class;  
}
```

y el archivo pd0010.pl sería...

```
#!/usr/bin/perl -w  
use CARRO;
```

```
$x = CARRO->new (50, "renault");  
$y = CARRO->new (70, "mazda");
```

```
$x->status; # imprime: velocidad=50  
$y->status; # imprime: velocidad=70
```

2 Clases de métodos

observe bien las 2 formas de llamar métodos de una clase:

1. CARRO->new(50, "renault") o new CARRO (50, "renault")

aquí el método "new" recibe 3

parámetros:

"CARRO", 50, "renault"

2. \$x->acelerar(7);

aquí el método "acelerar" recibe 2

parámetro:

\$x, 7

donde \$x es una referencia

a un objeto "CARRO"

cualquier método se puede llamarse de las 2 maneras... aunque lo usual

es que un método se llame con una de las 2 formas

1. cuando el método se llama como CARRO->new se dice que es un método de clase ósea que **no** esta asociada a un objeto específico aquí "new" es un método de clase.
2. cuando el método se llama como \$x->acelerar se dice que es un método de instancia ósea que esta asociada a un objeto aquí "acelerar" es un método de instancia también existen variables de clase y variables de instancia.

El módulo CGI

Este módulo se usa para leer los campos de una forma enviada desde el Netscape a nuestro programa Perl. a través de un servidor http como el Apache

```
# programa vt6100.pl
use CGI;
$q = new CGI;
# $q es una referencia tipo CGI...
# o mas simplemente un objeto CGI
$nom = $q->param ('nom');
$art = $q->param ('art');
$can = $q->param ('can');
# param es una función de CGI que nos da el valor de
un
# campo de la forma... 'nom' 'art' 'can' son nombres
# de campos de la forma en una página html que muy
# posiblemente salió de nuestro servidor http
# el pgma continúa revisando el pedido, aceptándolo
si esta ok
# y finalmente, dándole al cliente (con print por
supuesto)
# una respuesta adecuada...
```

La historia completa es esta:

Un cliente pide nuestra forma de pedidos digamos vt6100.html vt6100.html es algo como esto:

```
<h1>pedido</h1>
<form method=post action=http://epq.com.co/cgi-
bin/vt6100.pl>
<p>nombre <input name=nom size=30>
<p>codigo del articulo <input name=art size=8>
```

```

<p>cantidad<input name=can size=10>
<p><input type=submit value=enviar>
</form>

```

Una vez que el cliente llena la forma y da click en "enviar" el Netscape del cliente envía los campos de la forma al servidor. El servidor ejecuta el programa vt6100.pl (el programa Perl que se habló arriba) y le pasa los campos de la forma el programa vt6000.pl lee los campos de la forma usando el módulo CGI como se explicó arriba.

El módulo DBI

Este módulo se usa para acceder una base de datos como ORACLE bueno hay otro módulo involucrado, DBD::Oracle, pero eso es invisible ejemplo de una operación de consulta

```

use DBI;
use CGI;

$qry = new CGI;
$dbh = DBI->connect('dbi:Oracle:', 'useru', 'clave' );
# $dbh es un objeto de una clase
$ssth = $dbh->prepare ("select codemp, nomemp,
vinemp from emp
                                where ciaemp=?");
# esto hace que Oracle compile el "select"...
# $sth es un objeto de otra clase
$cia = $qry->param ( "cia");
# lea $cia de la forma del cliente
$ssth->execute ( $cia ) ;
# esto crea el cursor
while ( ($cod, $nom, $vin ) = $sth->fetchrow_array )
# aqui se lee el cursor
{

```

```

                                printf "%5s %30s %3s", $cod, $nom,
$vin ;
                                }
                                $sth->finish;
                                # esto cierra el cursor...
                                # parece que no es muy necesario... yo ya no los uso

```

ejemplo de una operación de actualización

```

                                use DBI;
                                $dbh = DBI->connect('dbi:Oracle:', 'usuario', 'clave' );
                                $sth = $dbh->prepare ("delete from emp where
ciaemp=?");
                                $sth->execute ( $cia ) ;
                                $dbh->commit;

```

el prepare y execute se pueden hacer simultáneamente con **do...**
pero "prepare" permite utilizar "placeholders" (las interrogaciones)
que se llenan en el "execute" información sobre el resultado de un
prepare o execute

```

$DBI::err
# número del error... análogo al sqlcode...
# es falso (no definido) si no hay error

```

```

$DBI::errstr
# texto del error
# es falso (no definido) si no hay error

```

información que se puede obtener después del execute

```

$DBI::rows
# nro de filas afectadas... puede ser 0
# no sirve en SELECT

```

información que se puede obtener después del execute de un

select

```
$sth->{NAME}
```

```
# referencia al arreglo de los nombres de la columnas
```

```
# se puede usar aunque no se seleccione ninguna
```

fila

```
$sth->{TYPE}
```

```
# referencia al arreglo de los tipos de los campos
```

```
$sth->{SCALE}
```

```
# referencia al arreglo de longitudes de los campos
```

El módulo LWP

Este módulo se usa para acceder servicios de internet como poner correo o leer una página... por supuesto que sin usar el Netscape

el módulo LWP maneja varios objetos:

LWP::UserAgent : el que se conecta al servidor

HTTP::Request : lo que se pide al servidor

HTTP::Response : lo que se recibe del servidor

ejemplo para enviar correo:

```
use LWP;
```

```
# 1. crear un agente ( user-agent )
```

```
$wuag = new LWP::UserAgent;
```

```
# 2. crear una petición ( request )
```

```
$wreq = new HTTP::Request (  
    POST => 'mailto:cjara@epq.com.co' );
```

```
# 3. llenar el encabezado (header) de la petición
```

```

$wreq->header (
    Subject => 'prueba de LWP' ,
    From    => 'alguien' );

# 4. llenar el contenido de la peticion
$wreq->content ( "me gusta este tutorial");

# 5. enviar la peticion con el agente
#   y obtener una respuesta "
$wres = $wuag->request ( $wreq );

# 6. examinar la respuesta
$wres->is_success ? print "exito \n": print "error \n";

```

Algo más... sobre archivos

```

open f1, "ps -xlw |";
# ejecutar el comando ps -xlw para leer su respuesta
# usando el descriptor f1

while ( <f1> )
{
    /httpd/ and print;
    # selecciona los demonios de httpd
}

```

el operador diamante sin descriptor, ósea <>, tiene un significado especial lee los archivos que se escribieron en la línea de comando

donde se ejecuto el programa Perl cuando se termina un archivo empata con el siguiente.

Es bueno aclarar que Perl coloca los argumentos que se dieron

en la línea de comando en el arreglo @ARGV...

si @ARGV no existe, "<>" lee del STDIN... o por supuesto el

archivo después del "<" en la línea de comando...

ósea la

redirección del STDIN...

como todo se puede en Perl... @ARGV puede ser alterado dentro

del programa...

```
@ARGV = ( "auxytd.98.3");
```

```
@a = <>;
```

```
# @a tiene todo el archivo
```

la función **read** para leer archivos binarios...

ósea archivos que no tienen líneas terminadas en "\n"

```
open FX, "auxytd.bin" ;
```

```
read FX, $buffer, 100;
```

```
# lee hasta 100 bytes
```

```
# retorna el numero de bytes leídos realmente
```

la función **select**

cambia el descriptor default de print y write

y devuelve el viejo descriptor

```
open ( F2, ">abc.dou");
```

```
$old = select ( F2 );
```

```
# $old contiene a STDOUT
```

```
print "abcdef";
```

```
# imprime en el archivo "abc.dou"
```

```
select ( $old );
```

```
# todo regresa a la normalidad
```

el descriptor **DATA** se refiere a todo lo que tiene el archivo del programa después de la línea `__END__` ...

antes de la línea `__END__` está el programa Perl...

por supuesto que la línea `__END__` es

opcional...

Perl también tiene los **here documents** de los shells de Unix...

los here documents se utilizan cuando se requiere un string de

muchas

líneas...

```
$x = << "ETX";
```

```
línea 1
```

```
línea 2
```

```
ETX
```

```
$x queda con: línea 1\nlínea2\n
```

```
ETX es cualquier palabra... $x queda con todas las
```

líneas antes

```
de la línea que tiene solo la palabra ETX...
```

```
ósea que si no se coloca ETX en una línea $x se
```

traga todo

```
el resto del archivo...
```

```
ojo! no olvide el ";" después de "ETX" en la primera
```

línea...

como se procesa un programa Perl

Perl procesa el programa en varias fases:

1. examina la línea `#!/usr/bin/perl...` buscando suiches...

este paso lo hace el shell de Unix pero en W95/NT lo hace el Perl

2. ejecución previa... un recuerdo de awk...
subrutinas BEGIN...
funciones "use" para cargar módulos
recuerde que los módulos empiezan por mayuscula...
esta ejecución temprana de "use" es otra diferencia con
"requiere" ... requiere se ejecuta en la fase 4

funciones "use" para dar directivas al compilador
en este caso la palabra que sigue a use empieza por minúscula.
use integer;
indica al compilador que solo queremos enteros
no integer;
indica al compilador que queremos números decimales

3. compilación... se había dicho que Perl no necesita compilar...
esta es una compilación a un código intermedio no a código de maquina... cancela si hay errores de sintaxis.

4. ejecución carnuda...
ejecuta el código intermedio... hasta encontrar **exit**

... o hasta que se acabe el archivo... o hasta que encuentre una línea con `__END__`

5. ejecución final...otro recuerdo de awk... subrutinas END... empezando por el último que se cargo

en las subrutinas BEGIN y END se puede omitir sub

```
BEGIN {  
    print "esto se escribe en la fase  
2\n";  
    print "aunque el programa tenga errores  
de sintaxis\n"  
}  
END {  
    print "esto se escribe en la fase 5\n";  
    print "después que el programa termina  
\n"  
}  
print "esto se escribe en la fase 4 \n";
```

La línea de comando de Perl

Perl se puede ejecutar con muchos suiches algunos de estos suiches también se pueden colocar en la línea "shebang" del programa la que empieza por `#!` la primera línea del shell podría ser

```
#!/usr/bin/perl -w
```

la línea de comando de Perl puede ser

```
perl -v
```

```
# da la version de Perl
```

```
perl -V
# muestra información sobre como se compilo el

Perl...

# al final muestra @INC que es otra e las variables
especiales

# de Perl

# @INC tiene los directorios donde Perl busca los
módulos

# y las librerias que se invocan con "use" y "requiere"

perl -w pd0010.pl
# muestra warnings... muy útil...

perl -d pd0010.pl
# puede correr el programa en cámara lenta...
# ejecuta el programa bajo el Perl degugger

perl -e '....'
# ejecuta las instrucciones entre comillas
# observe que no hay archivo de programa

perl -e '@a=(1..5); print "@a \n"'
# imprime: 1 2 3 4 5
```

el parámetro -e acompa#ado de otros parámetros se vuelve mas

útil

```
perl -n -e 's/pacho/francisco/; print' abc.txt
esto equivale al siguiente programa
```

```
while ( <> )
{
    s/pacho/francisco/;
```

```
        print;
    }
    donde <> toma archivos de @ARGV... en este caso
("abc.txt")
    ... recuerde el operador diamante
```

cambiando -n por -p nos ahorramos escribir el print
perl -p -e 's/pacho/francisco/' abc.txt
esto equivale al mismo programa

```
while ( <> )
{
    s/pacho/francisco/;
    print;
}
```

agregando -i podemos modificar el archivo abc.txt
perl -i.old -p -e 's/pacho/francisco/' abc.txt
remienda abc.txt y deja el archivo viejo como
abc.txt.old

Más sobre contexto

La función **wantarray** usada dentro de una subrutina
devuelve 1 si el llamador de la subrutina desea un
"contexto lista".

```
sub fun
{
    wantarray ? (1..5): 7;
}
$a = fun; # $a queda con
```

7

```
@a = fun; # @a queda con (1..5)
```

Herencia de objetos

La herencia es un concepto muy importante en la teoría de objetos...

se supone que uno debe hacer un objeto complicado heredando de otros

objetos mas simples...

p.e. si alguien ha creado ya la clase VEHICULO uno podría

empezar la clase CARRO diciendo que se deriva de VEHICULO...

ósea que las propiedades y métodos de VEHICULO se aprovechan

para definir la clase CARRO... o también que CARRO hereda de

VEHICULO... es una herencia espiritual... varias clases pueden

heredar de VEHICULO.

El arreglo @ISA le indica a Perl que si se invoca un método que no esta en el paquete lo puede buscar también en los paquetes mencionados en el arreglo @ISA... el ejemplo que sigue tiene 3 archivos.

1. archivo VEHICULO.pm

```
package VEHICULO;
sub funp1
{
    print "funp1 recibe @_";
}
return 1;
```

2. archivo CARRO.pm

```
package CARRO;
```

```

@ISA = ( VEHICULO );

sub func1
{
    $x = func1 ( "xxx" );
    # gracias a @ISA se invoca VEHICULO-
}
return 1;
>func1

```

3. archivo pd0010.pl

```

use CARRO;
package main;
CARRO->func1 (xxx) ;
# imprime: func1 recibe CARRO xxx
# aqui podemos decir que CARRO hereda de
VEHICULO...
# o que CARRO se deriva de
VEHICULO
# los archivos 1 y 2 deben colocarse en alguno de los
# directorios de @INC... o podemos poner un bloque
BEGIN que
# agregue nuestro directorio a @INC... recuerde perl
-V

BEGIN
{
    unshift @INC, "/appl/pd";
}

```

@ISA tiene un miembro automático llamado UNIVERSAL
 todo los módulo heredan automáticamente del módulo UNIVERSAL si
 CARRO invoca un método que no está en CARRO ni en VEHICULO

se busca en el paquete UNIVERSAL y si el método todavía no se encuentra se busca un método comodín llamado AUTOLOAD primero en CARRO luego en VEHICULO y finalmente en UNIVERSAL.