LENGUAJES DE CONTROL DE ÓRDENES: EL Bourne-Again Shell (BASH)

CONTENIDOS:

1. Introducción
2. Conceptos básicos
3. Variables y parámetros
4. Subshells
5. Operaciones aritméticas
6. Evaluación de expresiones lógicas
7. Escritura de programas
8. Sentencias de control
9. Funciones.
10. Operaciones de Entrada/Salida

1. Introducción

- El Bourne-Again SHell (BASH) es un lenguaje de control y programación de tareas interactivo para entornos LINUX que:
  - Muy compatible con otros lenguajes de control y programación de comandos, por ejemplo el Bourne Shell (BSH) o Korn Shell (KSH)
  - Soporta facilidades de operaciones de Entrada/Salida.
  - Soporta tipos de datos y atributos.
  - Soporta vectores unidimensionales.
  - Soporta aritmética entera.
  - Proporciona facilidades para el manejo de cadenas de caracteres.
  - Soporta facilidades para el control de tareas.
  - Soporta funciones y “alias”
  - Su implementación se basa en estándar IEEE POSIX (IEEE Standard 1003.1) en lo referente a intérpretes de órdenes.

1. Introducción

- **Inicio de sesión.** cuando un usuario entra en el sistema se le ejecuta el intérprete de comandos que tiene declarado en el fichero `/etc/passwd`, por ejemplo:

  ```
  lara:IWed4QkoRZ1d:101:12::/home/lara:/bin/bash
  ```

- **Cambio de intérprete de comando.** Simplemente tecleando el nombre, por ejemplo:

  ```
  # ksh
  ```

- **Retorno al intérprete de comandos original.** Simplemente tecleando lo siguiente

  ```
  # Ctl-d
  ```

1. Introducción

- **Invocación separada de varios intérpretes de comando.** Cada intérprete de órdenes posee un conjunto de especificaciones de entorno (variables de entorno) que se definen para cada usuario. En el caso del BASH, las variables se define leyendo **en orden** los archivos siguientes (si existen):
  1. `/etc/profile`
  2. `~/.profile`
  3. `~/.profile_login`

2. Conceptos básicos

- **Orden.** Se trata de una cadena de caracteres, organizada en palabras, en el que el carácter blanco se utiliza como separador entre palabras y en el que la primera palabra se interpreta como el nombre de un fichero ejecutable que contiene un programa.

```
# echo HOLA
```

- **Continuación de orden.** Si una orden termina con el carácter "\", entonces la siguiente línea se interpreta como una continuación de la anterior.

```
#echo HOLA \
> JUAN
```

- **Múltiples órdenes.** En una misma línea se pueden especificar varias órdenes empleando el carácter ";".

```
# echo HOLA; echo JUAN; echo MARIA
```

2. Conceptos básicos

- **Ejecución de órdenes en segundo plano.** Toda orden que finalice con el carácter “&” el BASH la ejecutará, pero no esperará por su finalización, por tanto, mientras ésta se ejecuta se podrá ordenar la ejecución de otra.

```
# ls /usr &
```

- **Tuberías (pipes).** Se trata de un mecanismo que permite la comunicación entre órdenes, de forma que la salidas que unas producen (por el canal de salida 1) son las entradas de otras (por el canal de entrada 0). Para establecer una tubería de comunicación entre órdenes se ha de especificar el carácter “|”.

```
# ls | wc -l
```

2. Conceptos básicos

- **Retorno.** Cuando una orden (programa) se ejecuta ésta devuelve un valor entero, el valor 0 se suele interpretar como ejecución con éxito, un valor distinto de cero se suele interpretar como ejecución con error.

- **Ejecución condicional.** Si dos órdenes están separadas por "&&", si la primera devuelve el valor 0 entonces la segunda se ejecuta.

```
# ls temp && echo temp existe
```

- **Ejecución condicional.** Si dos órdenes están separadas por "||", si la primera devuelve el valor distinto de 0 entonces la segunda se ejecuta.

```
# ls temp || echo temp no existe
# ls temp && echo temp existe || echo temp no\
  >existe
```

2. Conceptos básicos

- **Agrupando comandos.** Los comandos encerrados con “{}” se ejecutan combinando sus salidas. Para que la construcción sea correcta debe haber un espacio en blanco después del símbolo “{“ y antes del símbolo “}”. Los comandos deberán ir separados por “;” y el último dela línea deberá ir terminado también por “;”.

```
# { echo El contenido de temp: ; cat temp ; } | nl
```

2. Conceptos básicos

- **Redireccionamiento de entrada/Salida.** Mediante los símbolos “>” y “>>” podemos redireccionar el canal de salida de un comando.

```
# echo Hola > prueba
# echo Hola >> prueba
# > tmp
```

2. Conceptos básicos

- **Redireccionamiento de entrada/Salida.** Mediante el símbolo “<” podemos redireccionar el canal de entrada de un comando.

```
# mail Juan Maria < mensaje
```

- **Cerrando los canales de entrada y salida.** Mediante los operadores “<&-” y “>&-” podemos cerrar los canales de entrada y salida respectivamente de un comando.

```
#cat temp | wc -l <&-
```

2. Conceptos básicos

- **Redireccionando canales.** Con el operador “>&n” podemos redireccionar la salida de un comando al archivo especificado por el descriptor n, por tanto “n” debe ser descriptor de archivo válido.

```
# echo Este mensaje va al canal de error >&2
# ls tmp t.out >&2 2>ls.salida_error
# { echo Esto va al canal de salida >&1 ; \
echo Esto va al canal de error >&2 ; }
# { echo Esto va al canal de salida >&1 ; \
echo Esto va al canal de error >&2 ; } > salida
```

2. Conceptos básicos

- **Redireccionando canales.** Con el operador “n>&m”produce una salida donde los contenidos de los archivos referenciados por los descriptores n y m están anexados, por tanto “n” y “m”deben ser descriptores de archivos válidos.

```
# { echo Esto va al canal de salida >&1 ; \
echo Esto va al canal de error >&2 ; } >sal 2>&1
```

2. Conceptos básicos

- **Metacaracteres.** Mediante estos caracteres especiales podemos formar cadenas genéricas de caracteres:
  - “*” Cero o cualquier cadena de 1 o más caracteres
  - “?” Cualquier carácter
  - “[]” Cualquier carácter o rango de caracteres especificado
  - “!” Se usa con “[]” y significa negación, o sea, que no coincida con el carácter o rango de caracteres especificado.
  - “.” Por poseer un significado especial debe se especificado explícitamente.

2. Conceptos básicos

- **Ejemplos de uso de metacaracteres.**

```
# ls *ab*
# ls ??
#ls ???*
# ls [am]*[1-9]
# ls [!a]*
# ls .[a-h]
# rm *.[!ab]
```

2. Conceptos básicos

- **Operadores de patrones complejos.** El BASH permite buscar patrones formados por cadenas de caracteres:
  - * Cualquier cadena, incluida la cadena nula.
  - ? Un carácter.
  - […] Un carácter perteneciente a la lista.
  - ?(patrón) Cero o una ocurrencia de patrón.
  - *(patrón) Cero, una o más ocurrencias de patrón
  - +(patrón) Una o más ocurrencias de patrón
  - @(patrón) Una ocurrencia de patrón
  - !(patrón) Cualquiera excepto en las que aparezca patrón

- **Lista de patrones.** Se pueden especificar múltiples patrones utilizando el carácter “|” como separador.

2. Conceptos básicos

- **Ejemplos de uso de operadores con patrones complejos.**

```
*(A|i)
s?(?|??)
1?([0-9])
m+(iss)*
@([AC]la)*
```

2. Conceptos básicos

- **Ejecución de comandos.** Mediante la expresión $(comado) nos referimos al resultado de la ejecución de comando.

```
# echo La fecha de hoy es $(date)
# echo $(who -q) están en sesión
# echo Hay $(who | wc -l) usarios en sesión
```

- **Operaciones aritméticas.** Mediante la expresión $((expresion_aritmética)) podemos referirnos al resultado de la operación.

```
# echo $((8-3))
```

2. Conceptos básicos

- **El operador “ ~”.** Con este operador se referencia atributos de entorno referidos a pathnames, concretamente:
  - “~” sinónimo del contenido de la variable `HOME`
  - “~user” sinónimo del contenido de la variable `HOME` del usuario user
  - “~-” sinónimo del contenido de la variable `OLDPWD`
  - “~+” sinónimo del contenido de la variable `PWD`

2. Conceptos básicos

- **Alias.** Se trata de una facilidad soportada por el BASH mediante la cual podemos definir nuevos nombres de órdenes. Para usar esta facilidad disponemos de las siguientes órdenes:
  - ***alias*** *nuevo_nombre=orden*
  - ***alias***
  - ***unalias*** *nuevo_nombre*

- **Ejemplos:**

```
# alias p=echo
# p $Hola
# unalias p
# p $Hola
```

3. Variables y parámetros

- **Variable.** Un variable se define cuando se declara o cuando se le asigna un valor. Los nombres de variables deben empezar por un carácter alfabético (a-Z) al que le puede seguir cualquier carácter alfanumérico (a-Z,0-9). Existen variables cuyos nombres están compuesto por sólo número o caracteres especiales (!,@,#,%,*,?,$) que son de uso interno del BASH.

```
# X
# declare B
# Z=abc
# declare K1=hola
```

- **Acceso al valor de la variable.** Mediante el operador "$"

```
# echo $Z
# cd $HOME
```

3. Variables y parámetros

- **Atributos de variables.** Mediante la palabra clave “declare” podemos establecer valores y/o atributos a las variables.
  - ***declare*** *-atributo variable = valor*
  - ***declare*** *-atributo variable*

- **Atributos soportados por el BASH.**
  - ***declare*** *-i var*
  - ***declare*** *-l var*
  - ***declare*** *-a var*
  - ***declare*** *-A var*
  - ***declare*** *-r var*
  - ***declare*** *-f var*
  - ***declare*** *-t var*
  - ***declare*** *-u var*
  - ***declare*** *-x var*

3. Variables y parámetros

- **Desactivación de atributos.** Todos los atributos de una variable pueden ser desactivados, excepto el atributo de sólo lectura.
  - ***declare*** *+atributo variable*

- **Múltiples atributos.** Podemos establecer varios atributos de una variable mediante una sola línea de orden

```
$ declare -ix TMOUT = 300
```

- **Comprobación de los atributos de las variables.** Podemos listar conocer qué variables poseen un determinado atributo
  - ***declare*** *-atributo*
  - ***declare*** *+atributo*

3. Variables y parámetros

- **Asignación de valores a variables.** Ésta se puede realizar de distintas formas
  - *variable=literal* — `$ X=HOME`
  - *variable1=$variable2* — `$ X=$HOME`
  - *variable1=$(orden)* — `$ X=$(echo $HOME)`
  - *variable1=‘orden’* — `$ X='echo $HOME'`
  - *variable1=$(<archivo)* — `$ X=$(<prueba)`

- **Eliminación de variables.** Podemos eliminar definiciones de variables mediante la opción `unset`. Eliminar una opción no es lo mismo que asignar el valor nulo al contenido de una variable
  - ***unset*** *variable*

3. Variables y parámetros

- **Parámetros especiales.** El BASH hace uso de algunos parámetros de forma automática
  - **?** Código de salida de la última orden ejecutada
  - **$** Identificador de proceso del intérprete actual
  - **ERRNO** Código de error de la última llamada al sistema

- **Ejemplos:**

```
$ echo $?
$ echo $$
$ cat tmp.out
tmp.out: No such file or directory
$ echo $ERRNO
```

3. Variables y parámetros

- **Establecimiento de las variables de ambiente.** Cuando se invoca al BASH, éste, con el objeto de establecer el ambiente del usuario que le ha invocado, accede a los siguientes archivos en el orden que se indica:
    1. `/etc/profile`
    2. `~/.profile`
    3. `~/.profile_login`

3. Variables y parámetros

- **Algunas variables de ambiente:**

CDPATH	**IFS**	**TMOUT**
COLUMNS	**MAIL**	**VISUAL**
EDITOR	**MAILCHECK**	
ENV	**PATH**	
HISTFILE	**PS1, ..., PS4**	
HISTSIZE	**SHELL**	
HOME	**TERM**	

- Recomendación para evitar coincidencias con nombres de variables del ambiente: no usar nombres de variables que contengan sólo letras en mayúscula

3. Variables y parámetros

- **Expansión de variables.** El BASH soporta el acceso y modificación del contenido de las variables mediante un conjunto de operadores de expansión. Algunos ejemplos:
  - *$variable* `$ CA=hola; CA=$CAlifornia`
  - *${variable}* `$ CA=${CA}lifornia`
  - *${#variable}* `$ print ${#CA}`
  - *${variable:-literal}* `$ ${CA:-ab}`
  - *${variable:=literal}* `$ ${CA:=ab}`
  - *${variable:+literal}* `$ ${CA:+hola}`
  - *${variable:?}* `$ ${CA:?}`
  - *${variable:?literal}* `$ ${CA:?hola}`
  - *${variable#patrón}* `$ ${CA#ho}`
  - *${variable##patron}* `$ ${CA##la}`
  - *${variable%patrón}* `$ ${CA%ho}`
  - *${variable%%patrón}* `$ $(CA%%la}`

3. Variables y parámetros

- **Vectores de variables.** El BASH soporta vectores de dos tipos:
  - Indexados
  - Asociativos
- **Vectores indexados.** Los índices empiezan en 0.
  - ***declare*** *-a variable*
  - ***declare*** *variable[dim]* (aunque *dim* el BASH lo ignora)

- Ejemplos de inicializaciones válidas:

```
# vector=(11 12 13)
# vector[0]=11; vector[1]=12; vector[2]=13
```

- Se aumenta el tamaño del vector en dos elementos más y se inicializan estos dos nuevos elementos (posiciones 3 y 4)

```
# vector+=(14 15)
```

3. Variables y parámetros

- **Vector asociativo. Los índices son literales**
  - **declare -**A variable

- Ejemplos de inicializaciones válidas:

```
# dsemana=([lunes]=1 [martes]=2 [miércoles]=3)
# desemana[lunes]=1; dsemana[martes]=2; desemana[miércoles]=3
```

- Se aumenta el tamaño del vector en dos elementos más y se inicializan estos dos nuevos elementos (posiciones jueves y viernes)

```
# dsemana+=([jueves]=4 [viernes]=5)
```

3. Variables y parámetros

- **Acceso y modificación al contenido de vectores.**

  ${vector[n]}

  ${vector[*]}, ${vector[@]}

  ${#vector[*]}, ${#vector[@]}

  ${#vector[n]}

3. Variables y parámetros

- **Atributos.** Cómo las variables ordinarias, el BASH permite establecer atributos de vectores y éstos son los mismos que los definidos para las variables ordinarias. Los atributos se aplican a todos los elementos del vector.

```
# declare  -al vector
# declare –Au vector
```

3. Variables y parámetros

- **Las comillas simples**. Se utiliza para obviar el significado especial de los caracteres especiales ($,*,?,\,”,etc.) y realizar asignaciones que contienen espacios en blanco

```
# CA=‘ hola Juan’
# echo ‘$HOME’
```

- **Dobles comillas.** Igual que la comillas simples, excepto que no anulan el significado de los caracteres especiales: $, ` y \.

```
# CA=“$HOME”
```

- **La Comillas `** Se utilizan para asignar la salida producida por la ejecución de una orden

```
# CA=`date`
```

4. Subshells

- **Subshells.** Se trata de procesos hijos del intérprete actual y que ejecutan a otro intérprete de comandos. Éstos heredan las variables de entorno del intérprete padre y una vez finalizada su ejecución, las variable cosnervan el valor que tenían en el momento de la creación del subshell. Para generar un subshell se ha de utilizar la siguiente sintaxis:
  - ***(orden)***

- **Ejemplo:**

```
# PRUEBA=Hola
# (echo $PRUEBA)
# (PRUEBA=Adios; echo $PRUEBA )
# echo PRUEBA
```

5. Operaciones aritméticas

- **Especificación de operaciones aritméticas.** Cualquier operación de las soportadas por el BASH puede realizarse utilizando las siguientes sintaxis alternativas:
  - ***let "****operación-aritmética****"***
  - ***((*** *operación_aritmética****))***

  - **Ejemplos:** las siguientes expresiones son equivalentes

    # let "X=X+1"

    # ((X=X+1)

5. Operaciones aritméticas

- **Operadores aritméticos (por orden de precedencia).**
  - \- Evalúa el valor negativo de una expresión
  - ! Negación lógica
  - ~ Negación binaria
  - *, /, % Multiplicación, divisón, resto
  - +,- Suma, resta
  - <<, >> Desplazamiento a la izq., desplazamiento a la derch.
  - <=, < Menor que, menor que
  - >=, < Mayor o igual que, mayor que
  - == Igual que
  - != Distinto que
  - & And binario
  - ^ OR_exclusivo
  - | OR binario

5. Operaciones aritméticas

- **Operadores aritméticos (por orden de precedencia).**
  - && Operador relacional AND
  - || Operador relacional OR
  - = Asignación
  - *=, /=, %= multiplación y asignación, división y asignación, resto y asignación
  - +=, -= Suma y asignación, resta y asignación
  - <<=, >>= Desplazamiento izq. Y asignación, desplazamiento derecha y asignación
  - &=, ^=, |= And binario y asignación, OR-exclusivo y asignación, OR binario y asignación
  - (...) Especificación de operación y precedencia

6. Evaluación de expresiones lógicas

- **La orden [[...]] (que equivale a la orden ((…)))** Con ésta orden podemos evaluar expresiones condicionales e utilizar el resultado de dicha evaluación. Su sintaxis es:
  - *[[expresión_condicional]]*

  - **La orden [[...]] para operar con cadenas de caracteres.**
    - **[[-n** cadena **]]**
    - **[[-o** opcion **]]**
    - **[[-z** cadena **]]**
    - **[[** cadena1 = cadena2 **]]**
    - **[[** cadena1 **!=** cadena2 **]]**
    - **[[** cadena1 < cadena2 **]]**
    - **[[** cadena1 > cadena2 **]]**
    - **[[** cadena1 **=** patron **]]**
    - **[[** cadena1 **!=** patron **]]**

  - **La orden [[...]] para operar con enteros.**
    - **[[** expr1 **-eq** expr2 **]]**
    - **[[** expr1 **-ne** expr2 **]]**
    - **[[** expr1 **-lq** expr2 **]]**
    - **[[** expr1 **-lt** expr2 **]]**
    - **[[** expr1 **-ge** expr2 **]]**
    - **[[** expr1 **-gt** expr2 **]]**

6. Evaluación de expresiones lógicas

- **La orden [[...]] para operar con cadenas de caracteres.**
  - **[[-a** fich **]]**
  - **[[-d** dir **]]**
  - **[[-f** fich **]]**
  - **[[-L** fich **]]**
  - **[[-O** fich **]]**
  - **[[-G** fich **]]**
  - **[[-r** fich **]]**
  - **[[-s** fich **]]**
  - **[[-S** fich **]]**
  - **[[-u** fich **]]**
  - **[[-w** fich **]]**
  - **[[-x** fich **]]**
  - **[[** fich1 **-ef** fich2 **]]**
  - **[[** fich1 **-nt** fich2 **]]**
  - **[[** fich1 **-ot** fich2 **]]**

- **Expresiones complejas con la orden [[...]]**
  - **[[** expr1 **&&** expr2 **]]**
  - **[[** expr1 || expr2 **]]**
  - **[[!**expr **]]**
  - **[[(**expr**)]]**

7. Escritura de programas

- **Programas escritos en BASH.** El BASH puede ejecutar archivos ("scripts") que contienen ordenes, para ello el archivo tiene que tener permiso de ejecución.

```
# echo echo Mi primer programa > programa
# chmod 0755 programa
# ./programa
# bash programa
```

- **Parámetros posicionales ($0, $1, …, $N)** Son variables que se crean automáticamente y que contienen las distintas subcadenas que componen una orden.

```
# ptest HOLA JUAN
```

En este ejemplo, $0 es "ptest", $1 es HOLA y $2 es JUAN

7. Escritura de programas

- **Parámetros posicionales ($0, $1, …, $N)** Estos parámetros no admiten sentencias que impliquen la modificación de sus contenidos. La única manera de modificarlos es mediante la sentencia ***shift.***
- La orden ***shift*** desplaza a la izquierda los valores de los parámetros posicionales de la orden, a excepción de $0 que no le afecta. Se pierde el valor que tenía el primer argumento antes de la ejecución de ***shift.***
- Si ejecutamos:

  `# ./ptest HOLA JUAN`

  Los valores de $0, $1 y $2 serían respectivamente: `./ptest,` `HOLA` y `JUAN`

  Al ejecutar en el script la orden shift, entonces los valores que tendría $0, $1 y $2 serían: `./ptest,` `JUAN` y *`null`*`,` perdiéndose el valor `HOLA.`

7. Escritura de programas

- **Parámetros posicionales ($*, $@, $#)**
  - ***$**** Expande a la lista con los argumentos de la orden
  - ***$@*** Igual que $*
  - ***$#*** Expande al número de argumentos de la orden

7. Escritura de programas

Ejercicio: escriba un script llamado `ptest` con el siguiente contenido

```
#./bis/bash
echo "$0 $1 $2"
echo "$* $#"
shift
echo "$0 $1 $2"
echo "$* $#"
```

Ejecútelo así

```
# ./ptest enero febrero marzo abril mayo
```

¿Cómo afecta la orden `shift` a $1, $2, $3, $* y $#?

8. Sentencias de control

- **Sentencia “if”.**

  if *orden1*

  then

  Orden2

  else

  orden3

  fi

```
USERS=$(who | wc l)
if ((USERS == 1))
then
    echo “Hay 1 usuario“
else
    echo “Hay mas de 1 usuario“
fi
```

8. Sentencias de control

- **Sentencia “elif”.**

if *orden1*
then
    orden
elif *orden2*
then
    orden
elif *orden3*
then
    orden
else
    orden
fi

8. Sentencias de control

- **Ejemplo elif**

```
USERS=$(who | wc l)
if ((USERS == 1))
then
   echo “Hay 1 usuario“
elif ((USERS == 2))
then
   echo “Hay 2 usuarios“
else
   echo “Hay 3 o mas usuarios“
fi
```

8. Sentencias de control

- **Sentecia “while”.**

  while orden

  do

  orden1

  orden2

  .....

  done

- **Ejemplo**

```
while (($# != 0))
do
   echo $1
   shift
done
```

8. Sentencias de control

- **Sentecia “until”.**

  until *orden*

  do

  orden1

  orden2

  ....

  done

- **Ejemplo**

```
until ((($# == 0))
do
   echo $1
   shift
done
```

8. Sentencias de control

- **Sentencia “for”.**

  for *variable* ***in*** *palabra1 palabra2 .... palabraN*
  do
  orden
  done

- **Variante:**

  for *variable*
  do
  orden
  done

8. Sentencias de control

- **Ejemplo de sentencia “for”**

```
Itera=0
for X in A B C D
do
   echo “$Itera $X“
   ((Itera+=1))
done
```

8. Sentencias de control

- **Sentencia “case”.**

case *valor* ***in***

patron1) *orden*

orden;;

patron2) *orden*

orden;;

.....

patronN) *orden*

orden;;

esac

8. Sentencias de control

- **Ejemplo de sentencia case.**

```
case  $1 in
   @([a-z])) echo “carácter minúscula“;;
   @([A-Z])) echo “carácter mayúscula“;;
   @([1-9])([0-9])) echo “número entero“
esac
```

8. Sentencias de control

- **Sentecia “select”.**

  select *variable* ***in*** *palabra1 palabra2 palabra3 ... palabran*

  do

  orden

  done

- **Ejemplo**

```
select i in op1 op2 op3
do
   if [[ $i = op[1-3] ]]
   then
   echo “Ha seleccionado $REPLY: $i“
   fi
done
```

8. Sentencias de control

- **Sentecia “continue”.** Con esta sentencia forzamos la finalización de la iteración actual en bucles for, while y until.

```
while (($# != 0 ))
do
    if  [[ $1 = +([A-z]) ]]
    then
        echo “$1: argumento invalido“
        shift
    else
        echo “$1: argumento invalido“
        shift
        continue
    fi
done
```

8. sentencias de control

- **Sentecia “break”.** Con esta sentencia forzamos la salida de un bucle for, while, until. Admite dos formas

  break

  break *n*

```
for i in 1 2 3
do
    for j in 5 6
    do
        if (( i == 3 && j == 5 ))
        then
            break 2
        else
            printf $j
        fi
    done
done
```

9. Funciones

- Las funciones permiten agrupar órdenes, identificándolas con un nombre (que es el nombre de la función) y ejecutar estos grupos de órdenes mediante la invocación del nombre de la función.

function *name [()] { compound-commands } [redirections]*

- Antes de invocar a una función hay que definirla
- El paso de parámetros se hace por posición y se accede a ellos mediante $1, $2, …, $* en el cuerpo de la función.
- Las variables definidas dentro del cuerpo de la función son locales, a menos que estas se exporten al ambiente.
- Las variables definidas fuera del cuerpo de las funciones (variables globales) son accesibles a todas las funciones.

9. Funciones

- Una función puede devolver un valor de retorno de dos formas diferentes:
  - **Método 1**. Mediante la orden ***return valor.*** En este caso el código de retorno se obtiene mediante la orden ***$?***. Esta orden debe ser invocada antes de que se ordene la ejecución de cualquier orden en el script. En caso contrario $? devolverá el código se salida de la orden y no el de la función.

```
#!/bin/bash
 mifuncion() {
     return $(($1 + $2))
 }
 val1=10
 val2=30
 mifuncion $val1 $val2
 echo “Resultado= $?"
```

9. Funciones

- Una función puede devolver un valor de retorno de dos formas diferentes:
  - **Método 2.** Mediante la orden ***$(orden de salida por canal 1)***

```
#!/bin/bash
 mifuncion() {
     echo $(($1 + $2))
 }
val1=10
val2=30
valor=$(mifuncion $val1 $val2)
echo “Resultado= $valor"
```

10. Operaciones de entrada/salida

- Las operaciones de entrada/salida de una orden, incluidos los scripts, se realizan utilizando canales.
- Durante la ejecución de una orden, antes de utilizar un canal esté debe "abrirse" indicando si va a ser de entrada o/y salida.
- En la operación de apertura de un canal se le asigna un valor entero, llamado descriptor de fichero, que lo identifica de manera única.
- En un script los canales se abren y se cierran mediante la orden ***exec*.**

10. Operaciones de entrada/salida

- **Descriptores de ficheros:**
  - **0: Canal de entrada estándar.** Estándar significa que desde el inicio de la orden, este canal está abierto sin necesidad de abrirlo y además es de entrada. La norma es que por este canal se realizan operaciones de entrada de datos asociadas a la ejecución correcta de la orden.
  - **1: Canal de salida estándar.** Estándar significa que desde el inicio de la orden, este canal está abierto sin necesidad de abrirlo y además es de salida. La norma es que por este canal se realizan operaciones de salida de datos asociadas a la ejecución correcta de la orden.
  - **2: Canal de salida de error.** Estándar significa que desde el inicio de la orden, este canal está abierto sin necesidad de abrirlo y además es de salida. La norma es que por este canal se realizan operaciones de salida de mensajes asociadas a condiciones de error.
  - **3-9:** Descriptores disponibles para realizar lecturas/escrituras en ficheros.

10. Operaciones de entrada/salida

Ejemplo 1: ilustra la apertura y cierre de canales

```
#./bin /bash
# Este código ilustra como se abre un canal
exec 3 < ~/Entarda_Texto.txt
exec 4 > $PWD/Salida_Texto
exec 5 <> ../Entrada_Salida_Texto.txt
# Este código ilustra cómo se cierran canales
exec 3<&-
exec 4>&-
exec 5<>&-
```

10. Operaciones de entrada/salida

- Operación de entrada (lectura) de datos:

read [-ers] [-a aname] [-d delim] [-i text] [-n nchars] [-N nchars]
[-p prompt] [-t timeout] **[-u fd] [name ...]**

- Consideraciones generales de la orden ***read:***
  - ***name*** representa nombres de variables.
  - Se leen palabras desde el canal especificado mediante el argumento "-u fd". Si no se especifica este argumento las lecturas se realizarán en el canal 0 (canal de entrada por defceto).
  - El delimitador de palabra son los caracteres sepadores declarados, que por defecto son el espacio en blanco y el tabulador.
  - La variable de ambiente **`IFS`** especifica los caracteres separadores.

10. Operaciones de entrada/salida

- Operación de salida (escritura) de datos:

  printf [-v var] format [arguments]

- Consideraciones generales de la orden printf***:***
  - ***-v:*** La escritura se realiza sobre la variable ***var*** en vez de por el canal 0.
  - ***format:*** formato de la salida de los datos.
  - Arguments: valores a escribir.

10. Operaciones de entrada/salida

Ejemplo 2: ilustra operaciones de lectura/escritura

```
#./bin /bash
declare c1 c2 c3 c4 c5 c6 c7
# Aperturas de canales
exec 3 < /etc/passwd
# Para las lecturas
# Declaramos que el carácter separador es “:”
IFS=:
While read –u3 c1 c2 c3 c4 c5 c6 c7
do
    printf “USUARIO: %s   HOME: %s\n” $c1 $c2
done
```

10. Operaciones de entrada/salida

Ejemplo 3: ilustra operaciones de lectura/escritura

```
#./bin /bash
declare c1 c2 c3 c4 c5 c6 c7
# Aperturas del canal de entrada 3 asociado a /etc/passwd
exec 3</etc/passwd
# Para las lecturas
# Declaramos que el carácter separador es “:”
IFS=:
# Apertura canal de salida 4 asociado a ./users_homes.txt
exec 4>./users_homes.txt
# Para escribir en un canal distinto al 0
exec 1>&4
While read –u3 c1 c2 c3 c4 c5 c6 c7
do
    printf “USUARIO: %s   HOME: %s\n” $c1 $c2
done
```