

# Sesión 15: Lo que no he contado

Programación para Sistemas

---

Ángel Herranz

Otoño 2018

Universidad Politécnica de Madrid

# En capítulos anteriores. . .

**Sesión 11:** Contacto Bash

**Sesión 12:** Practicando Bash

**Sesión 13:** Mandatos, argumentos, variables y un poquito de redirección

**Sesión 14:** *Scripts*

# En el capítulo de hoy...

- Encuesta
- + *Teoría*: expansión, redirección, funciones, control de flujo y operadores de control.
- + *Scripts*

# Expansión i

 man bash y busca la sección *EXPANSION*.

# Expansión i

Q man bash y busca la sección *EXPANSION*.

- Parameter Expansion:

`${parameter}`

- Parameter Expansion (variantes):

`${parameter:-word}`

`${parameter:=word}`

`${parameter:?word}`

`${parameter:+word}`

`${parameter:offset:length}`

`${#parameter}`

# Expansión ii

- Command Substitution:

`$(command)`

# Expansión ii

- Command Substitution:

`$(command)`

`'command'`

# Expansión ii

- Command Substitution:

`$(command)`

`'command'`

- No es necesario aprenderse el manual
- Pero es necesario aprender a usarlo



# Ejemplo de pregunta

En el manual de Bash, se puede leer la siguiente descripción sobre la expansión de variables (La construcción es en realidad más compleja pero estas líneas bastan para resolver el ejercicio):

```
${!prefix*}
```

Nombres que encajan con prefijo. Expande a los nombre de variables que empiezan por prefix separados por espacios y ordenados por orden alfabético.

**Se pide** escribir las cuatro líneas de la salida estándar resultado de la ejecución de los siguientes mandatos Bash, suponiendo que no hay otras variables definidas que empiecen por “MIV”:

```
MIVUNO=
```

```
MIVDOS=
```

```
MIVTRES=
```

```
MIVCUATRO=
```

```
MIVCINCO=
```

```
echo ${!MIV*}
```

```
echo ${!MIVC*}
```

```
echo ${!MIVT*}
```

```
echo ${!MIVU*}
```

# Redirección

 `man bash` y busca la sección *REDIRECTION*.

# Redirección

Q man bash y busca la sección *REDIRECTION*.

- Básico:

```
command < file
```

```
command > file
```

```
command >> file
```

```
command 2> file
```

# Redirección

🔍 `man bash` y busca la sección *REDIRECTION*.

- Básico:

```
command < file
```

```
command > file
```

```
command >> file
```

```
command 2> file
```

- Redirecciones interesantes:

```
command 1>&2
```

```
command > file 2>&1  $\equiv$  command &> file
```

# Redirección

Q man bash y busca la sección *REDIRECTION*.

- Básico:

```
command < file
```

```
command > file
```

```
command >> file
```

```
command 2> file
```

- Redirecciones interesantes:

```
command 1>&2
```

```
command > file 2>&1  $\equiv$  command &> file
```

- Más: *here documents, duplicating file descriptors, etc.*

# Funciones

- Declaración

```
function f() {  
  ...  
}
```

```
function f {  
  ...  
}
```

```
f() {  
  ...  
}
```

- Uso: igual que cualquier programa

f arg1 arg2 arg3 ...

⚠ Cuidado con *tapar* un programar (imagina que llamas `ls` a una función)


# Funciones: parámetros y return

- **Parámetros**: son implícitos

```
function f() {  
    echo "Num args = $#"  
    echo "Arg 0 = $0"  
    echo "Arg 1 = $1"  
    ...  
}
```

- **return**: se refleja en el exit status

```
function f() {  
    return 1;  
}  
f  
echo $?
```

 man bash y buscar **shift**



# shift

 man bash y buscar **shift**

`shift [n]`

The positional parameters from `n+1 ...` are renamed to `$1 ....` Parameters represented by the numbers `$#` down to `$#-n+1` are unset. `n` must be a non-negative number less than or equal to `$#`. If `n` is 0, no parameters are changed. If `n` is not given, it is assumed to be 1. If `n` is greater than `$#`, the positional parameters are not changed. The return status is greater than zero if `n` is greater than `$#` or less than zero; otherwise 0.

# Funciones: ámbito

- **Parámetros**: no se hacen explícitos  
     `$#, $0, $1, $2, ...`
- Variables **globales** por defecto
- Pero se pueden **definir locales**

```
X=2
function f() {
    X=1
    echo $X
    ...
}
f
echo $X
```

```
X=2
function f() {
    local X=1
    echo $X
    ...
}
f
echo $X
```

# Control de flujo i

- if

`if command; then command; else command; fi`

$\equiv$

`if list; then list; else list; fi`

- for

`for name in words: do list; done`

- Muchas otras estructuras: **case**, **while**, etc.

# Control de flujo ii

- Operadores de control

`command & command`

`command ; command`

`command && command`

`command || command`

# Control de flujo ii

- Operadores de control

`command & command`

`command ; command`

`command && command`

`command || command`

```
command1 && command2
```

```
command1 || exit 1  
command2
```

```
if command1; then command2; fi
```

```
if command1; then  
    command2;  
else  
    exit 1;  
fi
```

# Control de flujo iii

- El programa **test**: man **test**
- Comprobaciones sobre **ficheros y strings**:

## SYNOPSIS

`test EXPRESSION`

...

`EXPRESSION` sets exit status. It is one of:

...

`-n STRING`

the length of `STRING` is nonzero

`STRING1 = STRING2`

the strings are equal


...

`-e FILE`

`FILE` exists

...

# Control de flujo iv

 Ejecuta `which [`

 ¿Qué es `which [?`

 Ejecuta

```
$ [
```

```
$ [ ]
```

```
$ [ -n "Hola"]
```


```
$ [ -z "Hola"]
```

```
$ test -n "Hola"
```

```
$ test -z "Hola"
```

 ¿Qué está pasando?

# Control de flujo iv

 Ejecuta `which [`

 ¿Qué es `which [?`

 Ejecuta

```
$ [
```

```
$ [ ]
```

```
$ [ -n "Hola"]
```

```
$ [ -z "Hola"]
```

```
$ test -n "Hola"
```

```
$ test -z "Hola"
```

 ¿Qué está pasando?

```
$ if [ -e $VIDEO ]; echo "$VIDEO existe"; fi
```