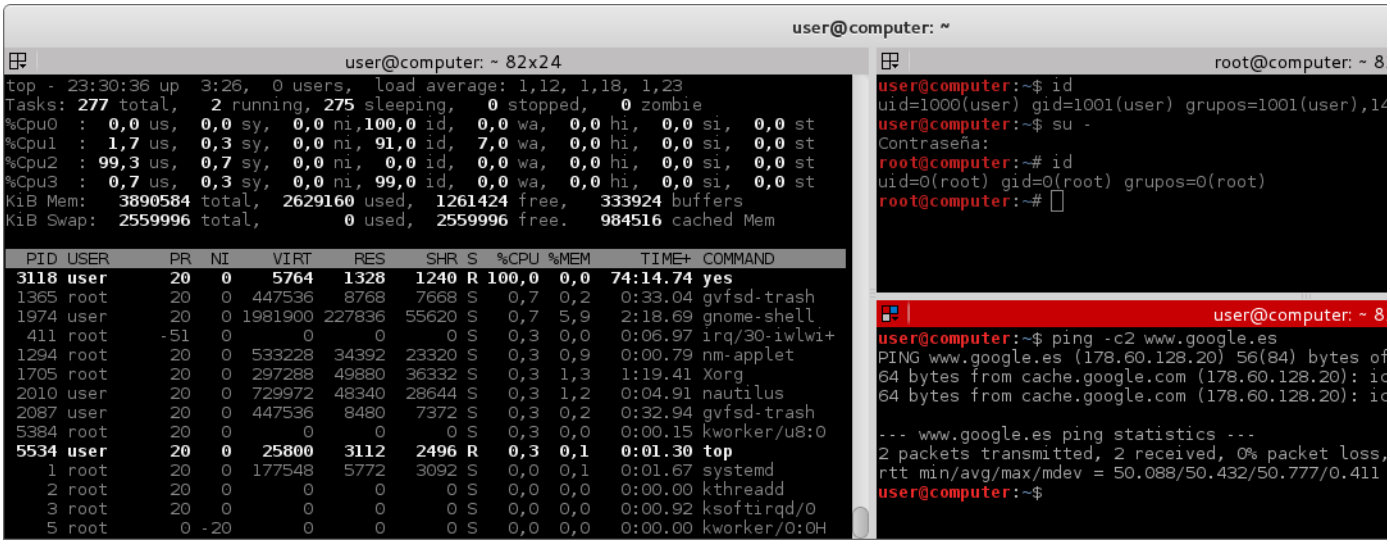
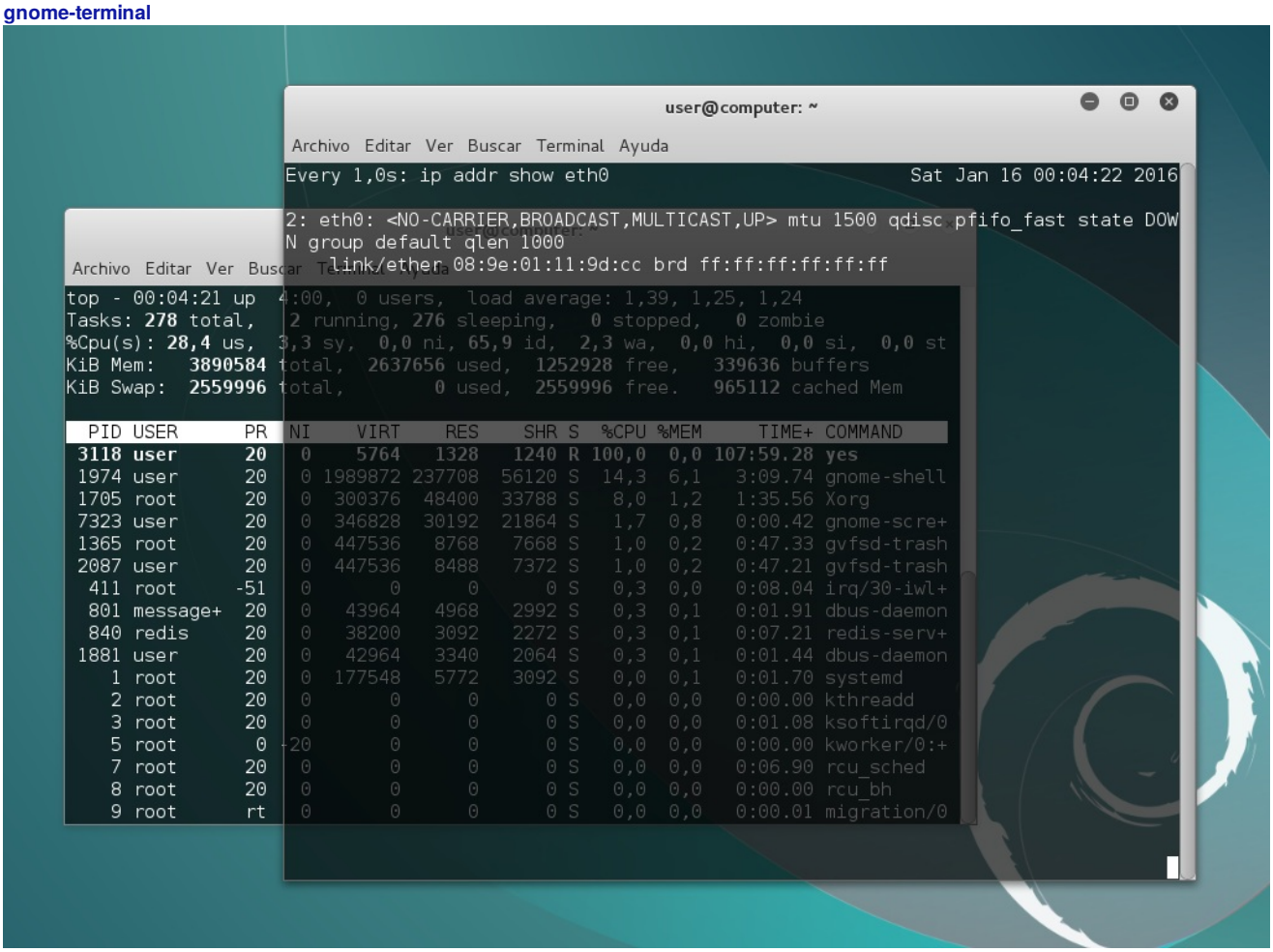
Comandos GNU/Linux e SHELL BASH (/bin/bash)

terminator



gnome-terminal

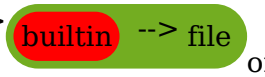


Ricardo Feijoo Costa

Variables e SHELL BASH: declare, typeset, readonly, set, unset, source, export, global, local, env, (), execución e subshell

echo $SHELL

- Orde de preferencia: 5 tipos

  alias -->keywords -->functions --> 

onde:
  alias --> $HOME/.bashrc, /etc/bashrc
  keywords --> palabras clave: function, if, for...
  functions --> funcións: nome_funcion() {...}
  builtin --> comandos internos: cd, type... Son internos a Bash e están sempre cargados na súa memoria.
  file --> scripts e programas executables (según PATH)

Práctica

- type <nome_comando>: amosa información sobre o tipo de comando
- type type: amosa información sobre o comando type
- type -t : amosa o tipo do/s comando/s pasado/s como argumento/s
  ```
  $ type -t declare
  builtin
  $ type -t typeset
  builtin
  ```
- type -a: amosa tódolos valores nos tipos do/s comando/s pasado/s como argumento/s, é dicir, tódalas posibilidades de execución do/s comando/s.
  ```
  $ type -a declare
  declare es una orden interna del shell
  $ type -a typeset
  typeset es una orden interna del shell
  ```
- help: Amosa a axuda dos comandos internos.
  ```
  $ help declare #amosa información sobre o comando interno declare
  ```
- man bash-builtins: Amosa a axuda de todos os comandos internos (builtins)
  ```
  # apt-get search ^manpages-es
  manpages-es-extra - Spanish extra manpages
  manpages-es - páginas de manual en español
  # apt-get -y install manpages-es manpages-es-extra
  ```

Práctica variables

declare

- **declare** e **typeset** son sinónimos. Permiten declarar variables.

  $ declare #Amosa todas as variables declaradas

  $ declare -p #Amosa todas as variables declaradas e cómo están declaradas

  $ declare AAA='aes' #Declara a variable de nome **AAA** co valor **aes**

  $ BBB='bes' #Declara a variable de nome **BBB** co valor **bes**

  $ BBBb1="$BBB" #Declara a variable de nome **BBBb1** co valor **bes**

  $ BBBb2=`echo $SHELL` #Declara a variable de nome **BBBb2** co valor **a saída do comando echo $SHELL**, esto é, co valor **/bin/bash**

  $ BBBb2=$(echo $SHELL) #Equivale ao comando anterior

  $ declare -p | egrep 'AAA|BBB' #Amosa o valor das variables **AAA**, **BBB**, **BBBb1**, **BBBb2** e como están declaradas, non existindo diferencia nas opcións de declaración, é dicir, por defecto, se non se emprega o comando interno **declare** para declarar as variables é como empregar o comando **declare *sen opcións***

  $ declare -r CCC='ces' #Declara a variable de nome **CCC** co valor **ces** en modo lectura, non podendo modificar o seu valor en sucesivas declaracións ou anular a súa definición co comando **unset**, é dicir, o seu valor non é modificable durante a vida da shell

  $ declare CCC='modificarces' #Amosa erro xa que a variable foi declarada en modo lectura e polo tanto non se permite asignar outro valor

  $ declare CCC='ces' #Amosa erro xa que a variable foi declarada en modo lectura e polo tanto non se permite asignar ningún valor, aínda que sexa o mesmo que se definiu co comando **declare -r**

readonly (declare -r)

- **readonly** e **declare -r** son sinónimos. Permiten declarar variables en modo lectura. Os valores non son modificables durante a vida da shell.

  $ readonly #Amosa o valor das variables declaradas en modo lectura

  $ readonly -p #Equivale ao comando anterior

  $ readonly DDD='des' #Equivale ao comando **declare -r DDD='des'**, polo que, declara a variable de nome **DDD** co valor **des** en modo lectura, non podendo modificar o seu valor en sucesivas declaracións ou anular a súa definición co comando **unset**

  $ declare -p | egrep 'CCC|DDD' #Amosa o valor das variables **CCC**, **DDD** e como están declaradas, non existindo diferencia nas opcións de declaración, demostrando que é o mesmo empregar **declare -r** que **readonly**

  $ readonly -p | egrep 'CCC|DDD' #Amosa o valor das variables **CCC**, **DDD** e como están declaradas, non existindo diferencia nas opcións de declaración, demostrando que é o mesmo empregar **declare -r** que **readonly**

set

- **set**. Sen opcións, amosa nomes e valores de cada variables da shell nun formato que pode ser reutilizado como entrada. Coa opción **-o** ou **+o** permite respectivamente establecer ou quitar atributos da shell.

  $ set -o #Amosa todas as opcións activas**(on)** e non activas**(off)** na shell

  $ set +o #Amosa todas as opcións activas**(-o)** e non activas**(+o)** na shell, co formato das ordes empregadas para conseguir ese resultado

  $ set -o noclobber #Con esta opción a shell bash non pode sobrescribir un ficheiro existente cos operadores de redirección (>, >&, <>). Engade a opción **noclobber** á variable **SHELLOPTS**, sendo esta variable un lista de elementos separados por dous puntos, de opcións activas da shell.

  $ set +o noclobber #Desactiva o comando anterior, é dicir, actívase a posibilidade de sobrescribir ficheiros cos operadores de redirección

  $ set -C #Equivale a **set -o noclobber**

  $ set +C #Equivale a **set +o noclobber**

  $ set -o noclobber

  $ set | grep noclobber

  $ echo $SHELLOPTS

  $ declare -p | grep SHELLOPTS

  $ set +o noclobber

  $ set | grep noclobber

  $ echo $SHELLOPTS

  $ declare -p | grep SHELLOPTS

  $ set -o noexec #Con esta opción a shell bash lee comandos pero non os executa. Emprégase para comprobar se os scripts teñen erros de sintaxe, pero para shells interactivos non ten efecto.

  $ set -n #Equivale ao comando anterior

  $ echo 'if [1 -eq 1];then echo igual' > /tmp/proba.sh

  $ bash -n /tmp/proba.sh #Comprobación da sintaxe do script /tmp/proba.sh

  $ set -o xtrace #Con esta opción a shell bash amosa o valor expandido de PS4, seguido do comando e os seus argumentos expandidos. Engade a opción **xtrace** á variable **SHELLOPTS**, sendo esta variable un lista de elementos separados por dous puntos, de opcións activas da shell.

  $ set -x #Equivale ao comando anterior

  $ set +o xtrace #Desactiva o comando **set -o xtrace**

  $ set +x #Equivale ao comando anterior

  $ set -o errexit #Con esta opción a shell bash sae inmediatamente se un comando remata a súa execución con erro. Hai que ter en conta que a shell non remata se o comando que falla é parte dun bucle **while** ou **until**, parte dunha sentencia **if**, parte dunha lista **&&** ou **||**, ou se o valor devolto polo comando invírtese mediante **!**
  Engade a opción **errexit** á variable **SHELLOPTS**, sendo esta variable un lista de elementos separados por dous puntos, de opcións activas da shell.

  $ set -e #Equivale ao comando anterior

$ set +o errexit #Desactiva o comando **set -o errexit**

$ set +e #Equivale ao comando anterior

unset

- **unset**. Elimina as variables da shell. Non pode eliminar as variables de lectura exclusiva (as definidas con **readonly** ou **declare -r**).

  $ EEE='ees' #Declara a variable de nome **EEE** co valor **ees**

  $ set | grep '^EEE=' #Amosa o valor da variable **EEE**

  $ echo $EEE #Amosa o valor da variable **EEE**

  $ unset EEE #Elimina a variable **EEE**

  $ echo $EEE #Amosa unha liña en branco debido ao comando **echo** xa que non existe a variable **EEE**

  $ set | grep '^EEE=' #Non amosa nada xa que non existe a variable **EEE**

source (.)

- **source** e o caracter punto **.** son sinónimos. Leen e executan os comandos existentes nun ficheiro dado na contorna actual da shell devolvendo o estado **$?** do último comando executado no ficheiro. O ficheiro non necesita ser executable e búscase primeiro nas rutas do **PATH** e logo na **ruta actual (pwd)** de execución do comando, a non ser que no nome dado exista unha barra inclinada **/**

  $ source $HOME/.bashrc #Recarga o ficheiro **$HOME/.bashrc** na shell actual.

  $ echo 'FFF="fes"' > variables.txt #Escribe **FFF="fes"** no ficheiro **variables.txt**, sobrescribindo este ficheiro no caso que exista.

  $ echo 'GGG="ges"' >> variables.txt #Engade **GGG="ges"** ao ficheiro **variables.txt**

  $ source variables.txt #Recarga o ficheiro **variables** na shell actual, buscando ese ficheiro nas rutas do **PATH** e na **ruta actual (pwd)** de execución do comando

  $ echo $FFF #Amosa o valor da variable **FFF**

  $ echo $GGG #Amosa o valor da variable **GGG**

  $ unset FFF && unset GGG #Elimina as variables **FFF** e **GGG** na shell actual

  $. variables.txt #Recarga o ficheiro **variables** na shell actual, buscando ese ficheiro nas rutas do **PATH** e na **ruta actual (pwd)** de execución do comando

  $ echo $FFF #Amosa o valor da variable **FFF**

  $ echo $GGG #Amosa o valor da variable **GGG**

  $ echo $PATH #Amosa o valor da variable **PATH**

  $ PATH=$PATH:/tmp #Engade a ruta **/tmp** á variable **PATH**

  $ echo $PATH #Amosa o valor da variable **PATH**

$ echo 'FFF="outrofes"' > /tmp/variables.txt #Escribe **FFF="outrofes"** no ficheiro **/tmp/variables.txt**, sobrescribindo este ficheiro no caso que exista.

$ echo 'GGG="outroges"' >> /tmp/variables.txt #Engade **GGG="outroges"** ao ficheiro **/tmp/variables.txt**

$ unset FFF && unset GGG #Elimina as variables **FFF** e **GGG** na shell actual

$ source variables.txt #Recarga o ficheiro **variables** na shell actual, buscando primeiro ese ficheiro nas rutas do **PATH** e logo na **ruta actual (pwd)** de execución do comando, de tal xeito que ao existir en **/tmp** e sendo **/tmp** unha ruta do **PATH**, os valores das variables **FFF** e **GGG** son **outrofes** e **outroges** respectivamente.

export

- **export**: Permite exportar variables á contorna actual da shell, de tal xeito que a partir da exportación son válidas na contorna actual da shell e en calquera subshell.

  Exportar unha variable nun script significa que quedará cargada en memoria de forma permanente ata que se peche a execución do script. Exportar variables ten unha gran importancia na execución dos scripts, pois moitas veces interésanos ter as variables cargadas en memoria durante toda a execución do script. É moi común crear un arquivo de variables a exportar para logo recollelo noutro arquivo do script mediante o comando **source**. A verdade é que podemos dicir que forman parella, case unha sen a outra non teñen moita razón de ser: **export-source**.

  $ export #Amosa unha lista de todas as variables exportadas na shell actual.

  $ export -p #Amosa unha lista de todas as variables exportadas na shell actual.

  $ export HHH='hes' #Declara unha variable de nome **HHH** co valor **hes** e ademais exporta a variable para que poida ser recoñecida na contorna actual da shell.

  $ echo $HHH #Amosa o valor da variable **HHH**

  $ declare -p | grep 'HHH=' #Amosa o valor da variable **HHH** e como está declarada

  $ export -p | grep 'HHH=' #Amosa que a variable **HHH** está exportada na shell actual.

  $ export -n HHH #Quita a propiedade exportación da variable de nome **HHH**

  $ declare -p | grep 'HHH=' #Amosa o valor da variable **HHH** e como está declarada

  $ export -p | grep 'HHH=' #Non amosa nada, xa que a variable **HHH** non está exportada na shell actual.

  $ declare -x HHH #Declara que a variable de nome **HHH** co valor **hes** ten a propiedade de exportación activa.

  $ declare -p | grep 'HHH=' #Amosa o valor da variable **HHH** e como está declarada

  $ bash #Executa unha subshell do usuario que executa o comando

  $ export -p | grep 'HHH=' #Amosa que a variable **HHH** está exportada na subshell actual.

  $ unset HHH #Elimina a variable **FFF** na subshell actual, pero segue existindo na shell pai

  $ exit #Sae da subshell actual e volta á shell pai

  $ export -p | grep 'HHH=' #Amosa que a variable **HHH** segue exportada na shell actual e mantén o seu valor.

global, local

- **global, local**:

  Todas as variables nos scripts bash, a non ser que se definan doutro xeito, son **globais**, é dicir, todas unha vez definidas poden ser empregadas en calquera parte do script. Para que unha variable sexa **local**, é dicir, soamente teña senso dentro dunha sección do script -coma nunha función- e non en todo o script debe ser precedida pola sentenza: **local**.

Por exemplo:

```
comezo sección
    local NOME=dentro
fin sección
```

onde, NOME soamente toma o valor **dentro** dentro da sección(función) onde se define. Realmente o exemplo anterior debería posuír unha función (ver o apartado [**Funcións**]).

```
dentro_variable_local() {
    local NOME=dentro
}
```

Quizás a execución dos seguintes exemplos, axuden:

Exemplo A: Sen a sentenza **local**

```
ONDE=script
echo $ONDE
dentro_variable_local() {
    ONDE=dentro
    echo $ONDE
}
dentro_variable_local
echo $ONDE
```

Exemplo B: Coa sentenza **local**

```
ONDE=script
echo $ONDE
dentro_variable_local() {
    local ONDE=dentro
    echo $ONDE
}
dentro_variable_local
echo $ONDE
```

Como se poder ver trala execución do **EXEMPLO A** obtemos a saída:

```
script
dentro
dentro
```

E trala execución do **EXEMPLO B** obtemos a saída:

```
script
dentro
script
```

Polo tanto conclúese que no **EXEMPLO A** ao non empregar a sentenza **local** a variable dentro da función é **global** e sobrescribe á variable definida antes da función.

env

- **env**: Executa un programa cunha contorna modificada según os parámetros cos que se execute, é dicir, executa un programa definindo que variables de contorna recoñece. Sen opcións ou nome de programa o comando amosa a contorna resultante (as variables globais da contorna), similar ao comando **printenv**

  $ type -t env
  file
  $ type -a env
  env is /usr/bin/env
  $ env #Amosa a contorna resultante. Igual que **printenv**
  $ env | sort #Amosa a contorna resultante ordeando alfabéticamente as variables da contorna.
  $ declare -p $(env | sort | cut -d'=' -f1 | grep -v '^_') #Amosa en orde alfabética como están declaradas as variables globais da contorna.
  $ env -i PATH=. ls #Executa o comando **ls** soamente se existe na ruta actual, xa que, coa opción **-i** comézase cunha contorna baleira, non existindo así variables da contorna e a maiores con **PATH=.** defínese a variable de contorna **PATH** co valor **ruta actual (punto)**
  $ env | grep LANG #Amosa a variable de contorna(global) **LANG**
  LANG=es_ES.UTF-8
  $ env man ls #Amosa a axuda do comando **ls** no idioma castellano (se están instaladas as manpages-es) porque na execución do comando anterior obtemos que **LANG=es_ES.UTF-8**
  $ env LANG=C man ls #Amosa a axuda do comando **ls** no idioma inglés porque con este comando estamos a modificar a variable de contorna **LANG** co valor **C**

Parénteses () ou subshell

- **( )** Se poñemos unha secuencia de ordes entre parénteses forzamos a estes comandos a ser executados nunha subshell

  $ (readonly AAA='aes'; echo $AAA; unset AAA; echo $AAA); AAA='bes'; echo $AAA; unset AAA; echo $AAA #Na subshell definida cos parénteses e na shell, as variables **AAA** son variables distintas.

Execución de scripts e subshell: bash, source(.) e permisos ugo(chmod +x)

- **shebang**: Tamén coñecido como *hashbang* ou *sha-bang* é unha convención en sistemas operativos tipo Unix que se utiliza nos scripts para indicar qué intérprete de comandos debe ser utilizado para executar o script. O shebang consiste nos caracteres ***#!*** seguidos da ruta ao intérprete. Por exemplo:
  1. En scripts bash: ***#!/bin/bash***
  2. En scripts python:
     #!/usr/bin/env python3
     #!/usr/bin/env python

- **chmod -x env.sh && bash env.sh**: Se executamos ***bash env.sh*** non é necesario ter permisos de execución no script e estamos a executar o script nunha subshell, co cal ao rematar o script eliminouse a subshell.

- **chmod +x env.sh && ./env.sh**: Se o **shebang** é ***#!/bin/bash*** e executamos mediante ***./env.sh***, sempre e cando o script teña permisos de execución, estamos a executar o script nunha subshell, co cal ao rematar o script eliminouse a subshell. É análogo á execución mediante o comando ***bash***.

- **chmod -x env.sh && . ./env.sh**: Se executamos mediante ***. ./env.sh*** ou ***source ./env.sh*** non é necesario ter permisos de execución e estamos a executar o script na shell actual.

Exemplos - Execución de scripts e subshell: bash, source(.) e permisos ugo(chmod +x)

- **Script env-exemplo1.sh**

  $ cat env-exemplo1.sh

```
#!/bin/bash #Liña necesaria para saber que shell executará o script(shebang)

env | sort | grep -v '^_' | tee env1.txt
```

1. Execución mediante: *bash env-exemplo1.sh*

   $ bash env-exemplo1.sh
   $ env | sort | grep -v '^_' | tee env2.txt
   $ diff env1.txt env2.txt
   30c30
   < SHLVL=1

   > SHLVL=0

Vemos que a variable **SHLVL** cambia de valor: de 0 na shell de traballo a 1 na shell do script. Isto é debido a que esta variable garda o valor da subshell coa que estamos a traballar, é dicir, indica o nivel de anidamento da shell. Así:

SHLVL=0 Significa que a variable definida por "env" toma o valor 0, é dicir, na saída do comando "env" estamos no nivel máis externo do shell, co cal non estás nunha shell anidada dentro doutra shell. É pouco común ver SHLVL establecido en 0, xa que normalmente sempre hai polo menos un nivel de shell en execución. Poderías ver SHLVL establecido en 0 se estás a revisar unha exportación de variables de entorno nun ficheiro ou se estás a revisar o entorno nun proceso moi cedo durante o arranque do sistema ou a inicialización do entorno de shell.

SHLVL=1 Significa que estamos nunha subshell e que a variable definida no entorno actual por "env" toma o valor 1

OLLO!: Se executamos na shell actual o comando:

```
$ echo $SHLVL

SHLVL=1
```

Vemos que obtemos o valor 1, iso é debido a que xa estamos a traballar nunha shell (a primeira dun posible anidamento).

Así, pechamos o terminal, abrimos outro e executamos:

```
$ env | grep SHLVL
SHLVL=0
$ echo $SHLVL #Amosa o valor para esta shell
1
$ bash #Accedemos a unha subshell
$ echo $SHLVL #Amosa o valor da subshell filla
2
$ bash #Accedemos a unha subshell
$ echo $SHLVL #Amosa o valor da subshell filla anterior, neta da primeira shell
3
$ exit #Sae da subshell e volta á shell pai correspondente
$ echo $SHLVL
2
$ exit #Sae da subshell e volta á shell pai correspondente
$ echo $SHLVL
1
$ exit #Pechamos o terminal
```

2. Execución mediante: *. ./env-exemplo1.sh || source ./env-exemplo1.sh*

```
$ . ./env-exemplo1.sh || source ./env-exemplo1.sh
$ env | sort | grep -v '^_' | tee env2.txt
$ diff env1.txt env2.txt
$
```

Agora NON existe cambio na variable **SHLVL** xa que o script non é executado nunha subshell senón na shell actual.
Isto é moi importante, xa que se executamos na shell actual o script, absolutamente "todo" o que fagamos no script afectará á shell actual, como o cambio de directorios e a exportación de variables.

- **Script env-exemplo2.sh**

user@debian:~$ cat env-exemplo2.sh

```
#!/bin/bash #Liña necesaria para saber que shell executará o script(shebang)

env | sort | grep -v '^_' | tee env1.txt
cd /tmp
```

1. Execución mediante: *bash env-exemplo2.sh*

```
user@debian:~$ bash env-exemplo2.sh
user@debian:~$ pwd
/home/user
user@debian:~$
```

A shell actual NON cambia de directorio xa que o script executouse nunha subshell

2. Execución mediante: *. ./env-exemplo2.sh || source ./env-exemplo2.sh*

```
user@debian:~$ . ./env-exemplo2.sh || source ./env-exemplo2.sh
user@debian:/tmp$ pwd
user@debian:/tmp$
```

A shell actual SI cambia de directorio xa que o script executouse na shell actual

- **Script env-exemplo3.sh**

$ cat env-exemplo3.sh

```
#!/bin/bash #Liña necesaria para saber que shell executará o script(shebang)

env | sort | grep -v '^_' | tee env1.txt
cd /tmp
declare -r LECTURA='read-only'
```

1. Execución mediante: *bash env-exemplo3.sh*

```
user@debian:~$ bash env-exemplo3.sh
user@debian:~$ declare -p | grep LECTURA
user@debian:~$
```

Na shell actual NON existe a variable LECTURA xa que o script executouse nunha subshell

2. Execución mediante: *. ./env-exemplo3.sh || source ./env-exemplo3.sh*

```
user@debian:~$ . ./env-exemplo3.sh || source ./env-exemplo3.sh
user@debian:/tmp$ declare -p | grep LECTURA
declare -r LECTURA="read-only"
```

Na shell actual SI existe a variable LECTURA xa que o script executouse na shell actual

EXECUCIÓN MEDIANTE PERMISOS UGO(chmod +x)

É análogo á execución mediante o comando ***bash***. Así:

```
user@debian:~$ ls -l env-exemplo?.sh
-rw-r--r-- 1 user user 54 abr 14 11:49 env-exemplo1.sh
-rw-r--r-- 1 user user 62 abr 14 11:49 env-exemplo2.sh
-rw-r--r-- 1 user user 93 abr 14 11:49 env-exemplo3.sh
user@debian:~$ chmod u+x env-exemplo?sh
-rwxr--r-- 1 user user 54 abr 14 11:49 env-exemplo1.sh
-rwxr--r-- 1 user user 62 abr 14 11:49 env-exemplo2.sh
-rwxr--r-- 1 user user 93 abr 14 11:49 env-exemplo3.sh
```

Executamos mediante permisos ugo:

```
user@debian:~$ ./env-exemplo1.sh
user@debian:~$ env | sort | grep -v '^_' | tee env2.txt
user@debian:~$ diff env1.txt env2.txt
30c30
< SHLVL=1
---
> SHLVL=0

user@debian:~$ ./env-exemplo2.sh
user@debian:~$ pwd
/home/user

user@debian:~$ ./env-exemplo3.sh
user@debian:~$ declare -p | grep LECTURA
user@debian:~$
```

CONCLUSIÓN: OLLO como se executa o script!!!

Ricardo Feijoo Costa

env: Exemplos entornos

LANG

```
$ echo $LANG
es_ES.UTF-8
```

LANG é unha variable de entorno que define o idioma predeterminado e as preferencias rexionais que se usan no sistema.

É unha variable clave en sistemas baseados en Unix e Linux que determina a configuración de idioma para os programas e as interfaces de usuario.

Cando un programa necesita mostrar mensaxes de texto, manexar caracteres especiais, ordenar listas ou realizar outras operacións relacionadas co idioma e a rexión, utiliza a configuración proporcionada pola variable LANG.

- **LANG=en_EN.UTF-8** establece o idioma predeterminado en inglés (inglés de Inglaterra) utilizando a codificación UTF-8. Isto significa que o sistema e as aplicacións mostraranse en inglés, seguindo as convencións e preferencias rexionais específicas de Inglaterra.
- **LANG=C** establece o idioma predeterminado en "C", que é un entorno estándar e básico sen localización, que adoita asociarse co inglés como idioma predeterminado. Utilízase comúnmente para aplicacións que non precisan soporte multilingüe ou para garantir un comportamento consistente e independente do idioma. Isto significa que o sistema e as aplicacións usarán as convencións predeterminadas do idioma "C", que poden diferir das convencións rexionais e lingüísticas específicas doutros idiomas.

DISPLAY

```
$ echo $DISPLAY
:0.0
```

DISPLAY é unha variable de entorno que indica a localización da pantalla no sistema X Window System (servidor gráfico) onde se mostrarán as aplicacións gráficas. Esta variable ten o formato **"hostname:display.screen"**. No caso de **":0.0"**, isto significa o seguinte:

- **"hostname"**: representa o nome do sistema onde se executa o servidor gráfico. No contexto de ":0.0", cando o valor está baleiro (como neste caso), significa que a aplicación se mostrará na mesma máquina onde se está a executar o servidor gráfico.
- **"display"**: refírese ao monitor físico ou á pantalla na que se mostrará a aplicación. Cada monitor identifícase cun número. Neste caso, o primeiro "0" representa o primeiro monitor dispoñible no sistema. Se tiveres máis dun monitor conectado ao sistema, poderías ter ":1.0", ":2.0", etc., para facer referencia aos outros monitores.
- **"screen"**: refírese ás pantallas virtuais dentro dun monitor físico. Nalgúns entornos de escritorio, como en XFCE, podes ter varias pantallas virtuais dentro dunha única pantalla física. Cada unha destas pantallas virtuais identifícase cun número. No contexto de ":0.0", o segundo "0" representa a primeira pantalla virtual na primeira pantalla física. Se tiveres máis dunha pantalla virtual na mesma pantalla física, poderías ter ":0.1", ":0.2", etc., para facer referencia ás outras pantallas virtuais.

Entón, **":0.0"** refírese á primeira pantalla na primeira pantalla do sistema local.

env

- **env**: Executa un programa cunha contorna modificada según os parámetros cos que se execute, é dicir, executa un programa definindo que variables de contorna recoñece. Sen opcións ou nome de programa o comando amosa a contorna resultante (as variables globais da contorna), similar ao comando **printenv**

Exemplos

env - Exemplo1

```
$ env | sort | grep -v '^_' | tee env1.txt # Lista todas as variables de entorno, ordeadas alfabeticamente, filtra as que non comezan por '_', e garda a saída no ficheiro env1.txt
$ echo $LANG # Imprime o valor da variable de entorno LANG, que indica o idioma e as preferencias rexionais
$ echo ñ # Imprime o carácter 'ñ'
$ echo $DISPLAY # Imprime o valor da variable de entorno DISPLAY, que indica a pantalla na que se mostrarán as aplicacións gráficas
$ xeyes & # Executa o programa gráfico xeyes en segundo plano
$ kill % # Detén a execución do programa gráfico xeyes
```

env - Exemplo2

```
$ env bash # Executa un novo shell (bash) coas mesmas variables de entorno que o shell actual
$ env | sort | grep -v '^_' | tee env2.txt # Lista todas as variables de entorno no novo shell e garda a saída no ficheiro env2.txt
$ diff env1.txt env2.txt # Compara as diferenzas entre os ficheiros env1.txt e env2.txt para verificar se hai cambios nas variables de entorno
$ echo $LANG # Imprime o valor da variable de entorno LANG no novo shell
$ echo ñ # Imprime o carácter 'ñ' no novo shell
$ echo $DISPLAY # Imprime o valor da variable de entorno DISPLAY no novo shell
$ xeyes & # Executa o programa gráfico xeyes en segundo plano dende o novo shell
$ kill % # Detén a execución do programa gráfico xeyes dende o novo shell
```

env - Exemplo3

```
$ env -i bash # Executa un novo shell (bash) cun entorno de variables de entorno limpo (sen herdar variables do shell actual)
$ env | sort | grep -v '^_' | tee env3.txt # Lista todas as variables de entorno no novo shell e garda a saída no ficheiro env3.txt
$ diff env2.txt env3.txt # Compara as diferenzas entre os ficheiros env2.txt e env3.txt para verificar se hai cambios nas variables de entorno
$ echo $LANG # Imprime o valor da variable de entorno LANG no novo shell
$ echo ñ # Tenta imprimir o carácter 'ñ' no novo shell pero non é posible escribir na consola o caracter 'ñ'
$ echo $DISPLAY # Imprime o valor da variable de entorno DISPLAY no novo shell, neste caso non existe.
$ xeyes & # Intenta executar o programa gráfico xeyes en segundo plano dende o novo shell pero non pode debido a que non existe a variable DISPLAY
```

env - Exemplo4

```
$ env LANG=C DISPLAY=:10.0 bash # Executa un novo shell (bash) coa variable de entorno LANG establecida en 'C' (inglés) e DISPLAY en ':10.0'
$ env | sort | grep -v '^_' | tee env4.txt # Lista todas as variables de entorno no novo shell e garda a saída no ficheiro env4.txt
$ diff env2.txt env4.txt # Compara as diferenzas entre os ficheiros env2.txt e env4.txt para verificar se hai cambios nas variables de entorno
$ diff env3.txt env4.txt # Compara as diferenzas entre os ficheiros env3.txt e env4.txt para verificar se hai cambios nas variables de entorno
$ echo $LANG # Imprime o valor da variable de entorno LANG no novo shell
$ echo ñ # Tenta imprimir o carácter 'ñ' no novo shell pero non é posible escribir na consola o caracter 'ñ'
$ echo $DISPLAY # Imprime o valor da variable de entorno DISPLAY no novo shell
$ xeyes & # Intenta executar o programa gráfico xeyes en segundo plano dende o novo shell pero non pode debido a que a variable DISPLAY define unha contorna gráfica non existente.
```

env - Exemplo5

```
$ env LANG=C DISPLAY=:0.0 bash # Executa un novo shell (bash) coa variable de entorno LANG establecida en 'C' (inglés) e DISPLAY en ':0.0'
$ env | sort | grep -v '^_' | tee env5.txt # Lista todas as variables de entorno no novo shell e garda a saída no ficheiro env5.txt
$ diff env2.txt env5.txt # Compara as diferenzas entre os ficheiros env2.txt e env5.txt para verificar se hai cambios nas variables de entorno
$ diff env3.txt env5.txt # Compara as diferenzas entre os ficheiros env3.txt e env5.txt para verificar se hai cambios nas variables de entorno
$ diff env4.txt env5.txt # Compara as diferenzas entre os ficheiros env4.txt e env5.txt para verificar se hai cambios nas variables de entorno
$ echo $LANG # Imprime o valor da variable de entorno LANG no novo shell
$ echo ñ # Tenta imprimir o carácter 'ñ' no novo shell pero non é posible escribir na consola o caracter 'ñ'
$ echo $DISPLAY # Imprime o valor da variable de entorno DISPLAY no novo shell
$ xeyes & # Executa o programa gráfico xeyes en segundo plano dende o novo shell
$ kill % # Detén a execución do programa gráfico xeyes
```

Ricardo Feijoo Costa

Corchetes e SHELL BASH: [, [[

echo $SHELL

- Orde de preferencia: 5 tipos

  alias -->keywords -->functions --> builtin --> file onde:
  alias --> $HOME/.bashrc, /etc/bashrc
  keywords --> palabras clave: function, if, for...
  functions --> funcións: nome_funcion() {...}
  builtin --> comandos internos: cd, type... Son internos a Bash e están sempre cargados na súa memoria.
  file --> scripts e programas executables (según PATH)

Práctica

- type <nome_comando>: amosa información sobre o tipo de comando
- type type: amosa información sobre o comando type
- type -t : amosa o tipo do/s comando/s pasado/s como argumento/s
  ```
  $ type -t [
  builtin
  $ type -t [[
  keyword
  ```
- type -a: amosa tódolos valores nos tipos do/s comando/s pasado/s como argumento/s, é dicir, tódalas posibilidades de execución do/s comando/s.
  ```
  $ type -a [
  [ es una orden interna del shell
  [ is /usr/bin/
  $ type -a [[
  [[ es una palabra clave del shell
  ```
- help: Amosa a axuda dos comandos internos, xa que o man non a posúe.
  ```
  $ help [ #amosa información sobre o comando interno [
  [: [ arg... ]
  Evalúa una expresión condicional.

  Este es un sinónimo para la orden interna "test", pero el último
  argumento debe ser un `]' literal, que coincida con el `[' inicial.
  ```

Corchetes dobres [[

A diferencia dos corchetes simples:

1. Non ten en conta o separador de campos IFS
2. Non ten en conta o globbing(expansión de caracteres)
3. Permite o operador =~ para comparar expresións regulares
4. Os operadores -o (OR) e -a (AND) empregados en corchetes simples non funcionan e son sustituidos respectivamente por || e &&

Práctica separador de campos IFS

$ rm -rf /tmp/corchetes #Eliminar /tmp/corchetes
$ mkdir /tmp/corchetes && cd /tmp/corchetes #Crear o cartafol /tmp/corchetes e acceder a este
$ VAR='sistema sistemas' #Declaramos e asignamos valor á variable VAR
$ if [$VAR = 'sistema sistemas']; then echo corchetes;fi
bash: [: demasiados argumentos
#Erro. Na comparación empregando **[** tense en conta o separador de campos IFS (espazo, tabulado, nova liña), polo cal ao sustituir o valor de VAR para comparar recolle 2 valores, e non pode realizar a comparación nunha soa sintaxe de if. É dicir:
if [$VAR =
é equivalente a
if test sistema sistemas
$ if [[$VAR = 'sistema sistemas']]; then echo corchetes;fi
corchetes
#Correcto. Empregando **[[** non se ten en conta o separador de campos IFS, polo cal ao sustituir o valor de VAR é un só campo (non ten en conta o espazo).
$ set | grep ^IFS #Ver valor variable separador de campos IFS (espazo, tabulado, nova liña)
IFS=$' \t\n'
$ oldIFS=$IFS #Gardamos o valor da variable IFS
$ IFS=$'\n' #Modificamos o valor da variable IFS para que o separador de campos sexa soamente 'nova liña'
$ if [$VAR = 'sistema sistemas']; then echo corchetes;fi
corchetes
#Correcto. Agora ao sustituir o valor de VAR para comparar non se ten en conta o espazo, xa que modificamos anteriormente a variable IFS.
$ if [[$VAR = 'sistema sistemas']]; then echo corchetes;fi
corchetes
#Correcto. Empregando **[[** non se ten en conta o separador de campos IFS, polo cal ao sustituir o valor de VAR é un só campo (non ten en conta o espazo).
$ IFS=$oldIFS #Recuperamos o valor por defecto da variable IFS

Práctica globbing(expansión de caracteres)

$ rm -rf /tmp/corchetes #Eliminar /tmp/corchetes
$ mkdir /tmp/corchetes && cd /tmp/corchetes #Crear o cartafol /tmp/corchetes e acceder a este
$ touch 's*' sistema sistemas #Creamos os ficheiros baleiros s*, sistema e sistemas
$ if [$(ls s*) = s*]; then echo OK;fi
bash: [: demasiados argumentos
#Erro. Na comparación empregando **[** faise globbing polo cal ao facer o comando **ls s*** para comparar recolle 3 valores, e como ten en conta o separador de campos IFS non pode realizar a comparación nunha soa sintaxe de if. É dicir:
if [$(ls s*) =
é equivalente a
if test s* sistema sistemas
$ if [[$(ls s*) = s*]]; then echo OK;fi
OK
#Correcto. Empregando **[[** compróbase se a execución do comando **ls s*** comeza polos caracteres **s***(agora s* é un patrón e non se expande) e non se ten en conta o separador de campos IFS, polo cal compárase se **s* sistema sistemas** comeza por **s***
$ set | grep ^IFS #Ver valor variable separador de campos IFS (espazo, tabulado, nova liña)
IFS=$' \t\n'
$ oldIFS=$IFS #Gardamos o valor da variable IFS
$ IFS=$'\n' #Modificamos o valor da variable IFS para que o separador de campos sexa soamente 'nova liña'
$ if [s* = s*]; then echo OK;fi
bash: [: demasiados argumentos
#Erro. Na comparación empregando **[**, ainda que agora o campo separador IFS non ten en conta os espazos e soamente en s* teriamos unha campo, faise globbing polo cal ao facer **s*** para comparar recolle 3 valores, e non pode realizar a

comparación nunha soa sintaxe de if. É dicir:
if [s* =
é equivalente a
if test s* sistema sistemas
$ if [[s* = s*]]; then echo OK;fi
OK
#Correcto. Empregando **[[** compróbase se **s*** é igual a **s***
$ IFS=$oldIFS #Recuperamos o valor por defecto da variable IFS

Práctica operador =~

$ if **[[** sistemas **=~** ^[sS].*[sS]$ **]]**; then echo OK;fi #O operador =~ compara a expresión regular ^[sS].*[sS]$, onde ^(comeza), [sS](calquera dos caracteres sS), .(calquera caracter), *(0 ou máis coincidencias da expresión regular anterior), [sS](calquera dos caracteres sS), $(remata)
OK

Práctica operadores ||(OR) e &&(AND)

$ if **[[** sistemas =~ ^[sS].*[sS]$ **||** SistemaS =~ ^[sS].*[sS]$ **]]**; then echo OK;fi #O operador =~ compara a expresión regular ^[sS].*[sS]$, onde ^(comeza), [sS](calquera dos caracteres sS), .(calquera caracter), *(0 ou máis coincidencias da expresión regular anterior), [sS](calquera dos caracteres sS), $(remata)
OK
$ if **[[** sistemas =~ ^[sS].*[sS]$ **&&** SistemaS =~ ^[sS].*[sS]$ **]]**; then echo OK;fi #O operador =~ compara a expresión regular ^[sS].*[sS]$, onde ^(comeza), [sS](calquera dos caracteres sS), .(calquera caracter), *(0 ou máis coincidencias da expresión regular anterior), [sS](calquera dos caracteres sS), $(remata)
OK

Corchete simple [

man test or man [

test CONDICION
equivale a
[CONDICION]
En sustitución do comando **test CONDICION** pódese empregar a expresión **[CONDICION]**

if then fi

if test CONDICION;then ...;fi
equivale a
if [CONDICION];then ...;fi
equivale a
[CONDICION] && comando
Debido ao **&&** executarase o comando cando se cumpra a condición.

En sustitución dos comandos **if then fi** pódese empregar a expresión:

[CONDICION] && comando

if then else fi

if test CONDICION;then ...;else ...;fi
equivale a
if [CONDICION];then ...;else ...;fi
equivale a
[CONDICION] && comando1 || comando2
Debido ao **&&** executarase o comando1 cando se cumpra a condición e debido ao **||** executarase o comando2 cando non se cumpra a condición.

En sustitución dos comandos **if then else fi** pódese empregar a expresión:

[CONDICION] && comando1 || comando2

Exemplos

```
#!/bin/bash

################################################################################
[ -f /etc/passwd ] && grep root /etc/passwd

if [ -f /etc/passwd ]; then
  grep root /etc/passwd
fi


################################################################################
[ -d $HOME ] && mkdir -p $HOME/bin/scripts

if [ -d $HOME ]; then
  mkdir -p $HOME/bin/scripts
fi


################################################################################
[ -r $HOME/bin/scripts/comprobar.sh ] && cat $HOME/bin/scripts/comprobar.sh

if [ -r $HOME/bin/scripts/comprobar.sh ]; then
  cat $HOME/bin/scripts/comprobar.sh
fi


################################################################################
[ -r $HOME/bin/scripts/comprobar.sh ] &&  [ -x $HOME/bin/scripts/comprobar.sh ] && .
$HOME/bin/scripts/comprobar.sh

if [ -r $HOME/bin/scripts/comprobar.sh ]; then
  if  [ -x $HOME/bin/scripts/comprobar.sh ]; then
    . $HOME/bin/scripts/comprobar.sh
  fi
fi


################################################################################
[ -r $HOME/bin/scripts/comprobar.sh ] || [ -x $HOME/bin/scripts/comprobar.sh ] && .
$HOME/bin/scripts/comprobar.sh

if [ -r $HOME/bin/scripts/comprobar.sh ] || [ -x $HOME/bin/scripts/comprobar.sh ]; then
    . $HOME/bin/scripts/comprobar.sh
fi


################################################################################
[ -r $HOME/bin/scripts/comprobar.sh ] && bash /bin/scripts/comprobar.sh || echo "Non se executa o
script"

if [ -r $HOME/bin/scripts/comprobar.sh ]; then
  bash $HOME/bin/scripts/comprobar.sh
else
  echo "Non se executa o script"
fi
```

Ricardo Feijoo Costa

while e read

Percorrer ficheiros

```
while read A
do
...
done < file
```

```
cat file | while read A
do
...
done
```

Scripts

```
#!/bin/bash

clear

echo '1 2 3 4 5' | while read A
do
  echo Resultado A: $A
  sleep 1
done

echo
echo '##############################'
echo

echo '1 2 3 4 5' | while read A B
do
  echo Resultado A: $A
  sleep 1
  echo Resultado B: $B
  sleep 1
done

echo
echo '##############################'
echo

echo '1 2 3 4 5' | while read A B C
do
  echo Resultado A: $A
  sleep 1
  echo Resultado B: $B
  sleep 1
  echo Resultado C: $C
  sleep 1
done

echo
echo '##############################'
echo

echo '1 2 3 4 5' | while read A B C D
do
  echo Resultado A: $A
  sleep 1
  echo Resultado B: $B
  sleep 1
  echo Resultado C: $C
  sleep 1
  echo Resultado D: $D
  sleep 1
done

echo
echo '##############################'
echo

echo '1 2 3 4 5' | while read A B C D E
do
  echo Resultado A: $A
  sleep 1
  echo Resultado B: $B
  sleep 1
```

```
63    echo Resultado C: $C
64    sleep 1
65    echo Resultado D: $D
66    sleep 1
67    echo Resultado E: $E
68    sleep 1
69 done
```

Ricardo Feijoo Costa

TRAP en SCRIPTS BASH

trap: Capturar sinais(nome ou número) e definir accións sobre estes nos scripts bash

$ trap 'comando' sinal1 [sinal2] ... [sinalN] #Capturar sinais e definir como acción un comando, unha lista de comandos ou unha función

$ trap '' sinal1 [sinal2] ... [sinalN] #Ignorar sinais

$ trap - sinal1 [sinal2] ... [sinalN] #Resetear sinais aos seus valores orixinais

Scripts

```
1 #!/bin/bash
2
3 #Atrapar sinais:
4
5 clear
6
7 ## Ignorar <Ctrl>+<c>: SIGGINT (2)
8 trap '' 2
9
10 echo '-------------------------------'
11 echo 'Ignorando <Ctrl>+<c>. Inténteo!'
12 echo '-------------------------------'
13 for i in $(seq 1 15)
14 do
15   echo $i
16   sleep 1
17 done
18
19
20 ## Resetear valor por defecto <Ctrl>+<c>: SIGGINT (2)
21 trap - 2
22
23 echo '---------------------------------------'
24 echo 'Devolvendo valor a <Ctrl>+<c>. Inténteo!'
25 echo '---------------------------------------'
26 for i in $(seq 1 15)
27 do
28   echo $i
29   sleep 1
30 done
```

```
1 #!/bin/bash
2
3 #Atrapar sinais:
4
5 ## <Ctrl>+<c>: SIGGINT (2)
6 trap 'clear;echo -e "\n<Ctrl>+<c>";id;sleep 2' 2
7
8 for i in $(seq 1 15)
9 do
10   echo $i
11   sleep 1
12 done
```

```bash
#!/bin/bash

##VARIABLES
cont=1

##FUNCIÓNS

#Atrapar sinal <Ctrl>+<c>: SIGGINT (2)
function control_c()
{
  echo -e "\n<Ctrl>+<c> ... Vai ser que non"
  cont=$(($cont+1))
}

#Atrapar sinal <Ctrl>+<z>: SIGTSTP (20)
function control_z()
{
  echo -e "\n<Ctrl>+<z> ... Vai ser que non"
  cont=$(($cont+1))
}


##main()
trap control_c SIGINT #Executa a función control_c como acción para o sinal SIGINT
trap control_z SIGTSTP #Executa a función control_z como acción para o sinal SIGSTP

while [ $cont -lt 3 ]
do
  read x
  echo $x
done
```

```bash
#!/bin/bash

#VARIABLES
cont=1

#FUNCIONS

##Atrapar sinais:
#### <Ctrl>+<c>: SIGGINT (2)
#### <Ctrl>+<z>: SIGTSTP (20)
control_c_z()
{
  echo -e "\n<Ctrl>+<c> ou <Ctrl>+<z>... Vai ser que non"
  cont=$((cont+1))
  if [ $cont -eq 3 ]; then
    VAR='sair'
  fi
}

trap control_c_z 2 20
while [ "$VAR" != 'sair' ]
do
  read x
  echo $x
done
```

```bash
#!/bin/bash

#VARIABLES
cont=1

#FUNCIONS

##Atrapar sinais:
#### <Ctrl>+<c>: SIGGINT (2)
#### <Ctrl>+<z>: SIGTSTP (20)
control_c_z()
{
  echo -e "\n<Ctrl>+<c> ou <Ctrl>+<z>... Vai ser que non"
  cont=$((cont+1))
  if [ $cont -eq 3 ]; then
    VAR='sair'
  fi
}

trap control_c_z 2 20
until [ "$VAR" == 'sair' ]
do
  read x
  echo $x
done
```

Ricardo Feijoo Costa

logger (syslog)

logger: Enviar mensaxes a syslog

$ logger -p prioridade mensaxe #Enviar a mensaxe a syslog coa prioridade (nivel facility) considerada.

$ logger -p prioridade mensaxe -t etiqueta #Enviar a mensaxe a syslog coa prioridade (nivel facility) considerada e etiquetada co texto etiqueta.

$ logger -p prioridade mensaxe -t etiqueta -s #Enviar a mensaxe a syslog coa prioridade (nivel facility) considerada e etiquetada co texto etiqueta, e ademais coa opción -s tamén amosa a mensaxe por pantalla no canal 2 (stderr).

syslog (man syslog)

facilidade.nivel acción
onde,
facilidade é o programa que xenera os logs
nivel é o nivel(prioridade) de logs
acción é o que se fai cos logs

Scripts

```
1 #!/bin/bash
2
3 #Atrapar sinais:
4
5 ## <Ctrl>+<c>: SIGGINT (2)
6 trap 'clear;echo -e "\n<Ctrl>+<c>";id;logger -p notice "Ctrl>+<c> executado";sleep 2' 2
7
8 for i in $(seq 1 15)
9 do
10   echo $i
11   sleep 1
12 done
```

Revisar syslog

```
# tail -f /var/log/syslog
...
Jan 22 13:47:02 computer user: <Ctrl>+<c> executado
Jan 22 13:47:05 computer user: <Ctrl>+<c> executado
Jan 22 13:47:09 computer user: <Ctrl>+<c> executado
```

```
1 #!/bin/bash
2
3 #Atrapar sinais:
4
5 #FUNCIÓNS:
6 function f_trap() {
7   clear
8   echo -e "\n<Ctrl>+<c>"
9   id
10  logger -p warn "<Ctrl>+<c> executado no script $0" -t WARNING!!!
11  logger -p warn "<Ctrl>+<c> executado no script $0" -t AVISO!!! -s
12  sleep 2
13 }
14
15
16 ## <Ctrl>+<c>: SIGGINT (2)
17 trap f_trap 2
18
19 for i in $(seq 1 15)
20 do
21   echo $i
22   sleep 1
23 done
```

Executar script logger2.sh

```
$ bash logger2.sh
...
<Ctrl>+<c>
uid=1000(user) gid=1001(user) grupos=1001(user)
AVISO!!!: <Ctrl>+<c> executado no script logger2.sh
...
```

Revisar syslog

```
# tail -n2 -f /var/log/syslog
...
Jan 22 13:48:12 computer WARNING!!!: <Ctrl>+<c> executado no script logger2.sh
Jan 22 13:48:12 computer AVISO!!!: <Ctrl>+<c> executado no script logger2.sh
Jan 22 13:48:15 computer WARNING!!!: <Ctrl>+<c> executado no script logger2.sh
Jan 22 13:48:15 computer AVISO!!!: <Ctrl>+<c> executado no script logger2.sh
```

Ricardo Feijoo Costa

Nomenclatura variables bash

En Bash, as variables poden definirse seguindo certas regras:

1. **Nomes de variables**:
   - SI poden conter letras (maiúsculas e minúsculas), números e guións baixos (_) pero NON poden comezar cun número.
   - NON están permitdos os caracteres especiais:
     1. Caracteres de puntuación: NON se poden utilizar en nomes de variables: !, ", #, $, %, &, ', (,), *, +, ,, -, ., /, :, ;, <, =, >, ?, @, [, \,], ^, `, {, |, } e ~
        - ! (signo de exclamación): Utilízase para invocar comandos do historial ou para negar expresións.
        - # (almofada): Utilízase para iniciar comentarios en Bash.
        - $ (signo de dólar): Utilízase para acceder ao valor dunha variable, para realizar expansións de variables e expresións aritméticas.
        - % (signo de porcentaxe): Utilízase para realizar operacións de módulo ou para realizar sustitucións e formatos de cadeas no contexto do comando *printf*.
        - & (ampersand): Utilízase para executar comandos en segundo plano ou para conectar comandos nun pipeline.
        - ' (comilla simple): Utilízase para protexer cadeas de texto da interpretación de metacaracteres.
        - ((paréntese esquerdo): Utilízase para agrupar comandos ou expresións.
        -) (paréntese dereito): Utilízase para pechar agrupacións de comandos ou expresións.
        - * (asterisco): Utilízase como comodín para coincidir con calquera secuencia de caracteres.
        - + (signo de máis): Utilízase para realizar operacións aritméticas ou en expresións regulares para representar a sustitución de 1 ou máis caracteres.
        - , (coma): Utilízase para separar elementos en listas ou argumentos.
        - - (guión): Utilízase para restar números e indicar opcións de comandos.
        - . (punto): Utilízase como primeiro caracter non nome dun ficheiro para indicar que está oculto, para separar extensións de ficheiros e en expresións regulares representa calquera caracter.
        - / (barra diagonal): Utilízase para separar directorios en rutas de ficheiros ou para indicar a división entre comandos.
        - : (dous puntos): Utilízase para indicar portos TCP, UDP en conexións de rede e separación de valores na variable PATH.
        - ; (punto e coma): Utilízase para finalizar un comando ou para separar sentenzas nun script.
        - < (menor que): Utilízase para realizar comparacións numéricas ou para redirixir a entrada dun comando.
        - = (signo de igual): Utilízase para asignar valores a variables ou para realizar comparacións.
        - > (maior que): Utilízase para realizar comparacións numéricas ou para redirixir a saída dun comando.
        - ? (signo de interrogación): Utilízase como comodín para coincidir cun só carácter ou para realizar tests de condición.
        - @ (arroba): Utilízase para acceder a arrays asociativos.
        - [(corchete esquerdo): Utilízase para agrupar expresións ou para iniciar estruturas condicionais.
        - \ (barra invertida): Utilízase para escapar caracteres especiais ou para separar comandos na liña de comandos.
        -] (corchete dereito): Utilízase para pechar agrupacións de expresións ou para finalizar estruturas condicionais.
        - ^ (circunflexo): Utilízase para negar patróns de comodíns ou para realizar operacións bit a bit.
        - ` (comilla grave): Utilízase para executar comandos como o shell. Análogo a $()
        - { (chave esquerda): Utilízase para iniciar bloques de código ou para definir estruturas de datos.
        - | (barra vertical): Utilízase para conectar comandos nun pipeline.
        - } (chave dereita): Utilízase para pechar bloques de código ou para definir estruturas de datos.
        - ~ (til): Utilízase para acceder ao directorio persoal do usuario ou para negar patróns de comodíns.

2. Caracteres de control:
   - Espazos en branco (\x20): NON están permitidos porque a shell os utiliza para separar comandos e argumentos.
   - Tabuladores (\t): NON están permitidos porque a shell os utiliza para a indentación do código e poderían xerar confusións ao interpretar a variable.
   - Retorno de carro (\r): NON están permitidos porque para a shell este carácter ten un significado especial na codificación de texto e non se pode utilizar en nomes de variables.
   - Salto de liña (\n): NON están permitidos porque na shell marcan o final dunha liña de texto.

2. **Convencións**:
   No que respecta a nomenclatura ou convención para definir variables en Bash, non hai unha regra estrita que deba seguirse en todos os casos, pero o uso de maiúsculas para distinguir as variables pode ser útil para mellorar a lexibilidade do código. Con todo, sempre é importante seguir as convencións de codificación específicas do proxecto ou do equipo.
   Podes atopar máis información sobre as variables de Bash na documentación oficial de Bash ou en páxinas de referencia sobre scripting en Bash en GNU/Linux. Por exemplo:
   - O manual de Bash:

     ```
     $ man bash
     ```

   - O manual de Referencia de Bash: **https://www.gnu.org/software/bash/manual/bash.html**
   - Convencións da comunidade: Os desenvolvedores de Bash e a comunidade de usuarios estableceron certas convencións non oficiais para a nomenclatura das variables. Estas convencións baseanse na claridade, lexibilidade e consistencia do código. Algúns destes puntos inclúen:
     - Nomes descritivos: O nome da variable debe reflectir claramente o contido que almacena.
     - Evitar nomes moi longos ou complexos: Os nomes curtos e claros son máis fáciles de entender.
     - Ser coherente co estilo: Se se usan maiúsculas nalgúns nomes, manter a coherencia en todo o código.
   - Guías de estilo: Existen guías de estilo de programación específicas para Bash que recomendan boas prácticas para a nomenclatura das variables. Un exemplo notable é a Guía de estilo de Google para Bash: **https://google.github.io/styleguide/shellguide.html**

Resumo

A nomenclatura das variables en Bash basease nunha combinación de convencións establecidas pola comunidade, boas prácticas de programación e, en algúns casos, guías de estilo específicas. Non existe unha definición oficial única, pero seguir estas recomendacións xerais axuda a crear código máis claro, lexible e doado de manter.

Lembra que as variables en Bash deben comezar cunha letra ou un guión baixo (_) e poden conter letras, números e guións baixos. É unha boa práctica utilizar nomes descritivos que reflictan o propósito da variable para facer o código máis lexible e mantible.

Ricardo Feijoo Costa