OPERATING SYSTEM: UNIX/LINUX

Course 3

Osman SALEM
Maître de conférences - HDR
osman.salem@parisdescartes.fr

1

Shell Programming: a basic script

- Write shell program
  - scripts containing a series of shell commands
- The first line of the script should start with
  - #!/bin/bash (She Bang: #!)
  - which indicates to the kernel the interpreter

```
#!/bin/sh
# fichier : bonjour.sh
# Display Bonjour for the user
echo "Bonjour $USER"
```

- bash$ **chmod +x bonjour.sh**
- bash$ **./bonjour.sh** (./ is used to start a program)
  - Bonjour dany
- She Bang: #! Must be the first line
- Comments
  - Every lines that start with **#** Unless the first line

2

A Basic Script

-
  ```
  pwd
  ls -C
  date
  ```
- To make a file executable, use the `chmod` program
  ```
  chmod +x myscript
  ```
- To run the file as a program, simply type:
  ```
  ./myscript
  ```
- If the directory that contains the script is in your `PATH`, this can be abbreviated further to:
  ```
  myscript
  ```

3

A Basic Script

- *Any* UNIX command may be added to a script
  - ./`myscript`
  - *export PATH=$PATH:. ; echo $PATH*
  - `Myscript`
- *Create bin directory in your home*
  - *mkdir bin*
  - *mv bonjour.sh bin*
  - *bonjour.sh*

4

The `echo` Command

- `echo` is a shell built-in command
- Its function is simple: to write its command-line parameters to Standard Output. If no parameters are given, a blank line (carriage-return) is output
- It is primarily used to display messages to the users of the script
- For example:

```
$ echo My name is Mark
My Name is Mark
$
```

5

The `read` Command

- `read` is a shell built-in command for reading from Standard Input (usually the keyboard) and storing the information in shell *variables*
- It is mostly used to receive the answers to questions and prompts issued by the script
- For example:

```
$ read name
Mark Virtue
$
```

- The shell variable `name` now contains the value `Mark Virtue` and can be examined by typing

```
$ echo $name
```

6

The `read` Command

- `read` can break the line of input into several variables, as follows:

```
$ read firstname surname
  Mark Virtue
  $
```

- The shell variable `firstname` contains the value `Mark` and `surname` contains the value `Virtue`
- Input is separated by spaces and tabs
- If more words are provided than there are variables, the extra words are added to the last variable
- If not enough words are provided, the extra variables will contain nothing

7

Read command

- Options

read –s	(silent)
read –n N	(takes only N character)
read –p "message"	(prompt message)
read –t T	(timeout T seconds)

8

command Substitution

- Substitution
  - $() or ` `
- Example :
  - echo "`whoami`, we are the `date` "
  - echo "$(whoami), we are the $(date)"
- Compare :
  - pwd
  - echo pwd
  - echo `pwd`
  - echo "there is `ls | wc -l` file(s) in `pwd` "
  - NOW=`date`
  - MYDIR=`pwd`

9

command Substitution

```
#!/bin/bash
# This script displays some information about your environment
        echo "Bonjour. Nous sommes le $(date)"
        echo "Votre répertoire du travail est: $(pwd)"
        echo 'Votre répertoire du travail est: $(pwd)'
```

- single quotation: prevents the shell interpretation of commands

```
$ echo $person
   max
$ echo "$person"
   max
echo "$LOGNAME needs $1000 in `date +%B`" (utiliser \$)
   xyz needs 000 in October
$ echo '$person'
   $person
$ echo \$person
   $person
```

10

Variables

- Nom des variables
  - May contains

    `a-z, A-Z, 0-9` et "_"
  - Must begin with a letter
  - Case sensitive
  - No space before and after afecting a value (=)
  - Use double quotation in case the string contains a space
  - Example:
    - `month=Janvier`
    - `street="Rue Saint Pères"`
    - `echo $street`
    - `unset street`
    - `echo $street`

11

Variables

- `PREF=counter`
- `WAY=${PREF}clockwise`
- `FAKE=${PREF}feit`
- `echo $WAY $FAKE`
- `person=`
- `echo $person`
- `unset person`

12

Variables

- Variables are used to store a value
  - files="notes.txt report.txt"
  - echo $files
  - A=10; echo $A; unset A; echo $A
- Environment variables
  - env: displays whole environment variables
  - export files="notes.txt report.txt"
  - Or files="notes.txt report.txt" ; export files
- PATH
  - echo $PATH
  - echo $USER
  - echo $PWD; echo $HOSTNAME; echo $PS1;
  - export PS1="[\u:\w]\$"

13

Shell Variables (cont.)

- (For experienced programmers: All shell variables are *strings*)
- Values may be assigned to a variable by use of the "=" sign, for example:

```
sport=basketball
```

- There must be no **spaces** on either side of the "="
- If you need to assign a value that contains spaces to a variable, use the " character. For example:

```
street="Smith Avenue"
```

- To retrieve the contents of a variable, use the "$" sign before the variable name:

```
echo You live on $street
```

14

Special Characters

- These special characters should be avoided when naming files
  - Note that it is *never* possible to give a file a name that includes the `/` character (although this character is not special to the shell)
- If it ever becomes necessary to pass one of these characters as a parameter to another program, one of three actions is required:
  - Prefix the character with a `\` (for example, `\$`)
  - Surround the character with a pair of " characters (for example `"#"`) Note, this doesn't work for all characters
  - Surround the character with a pair of ' characters (for example `'$'`) This works for all characters except `'`

15

Comments

- A *comment* is a piece of human-readable text added to a script to make the code more understandable
- A comment is any part of a line of a script that follows the `#` character
- For example:

```
 #  Count the number of users on the system
who | wc -l    # wc means "word count"
```

- Comments are an important part of software development – their use dramatically cuts down on maintenance time and costs
- You are strongly encouraged to comment *all* your code

16

Environment Variables

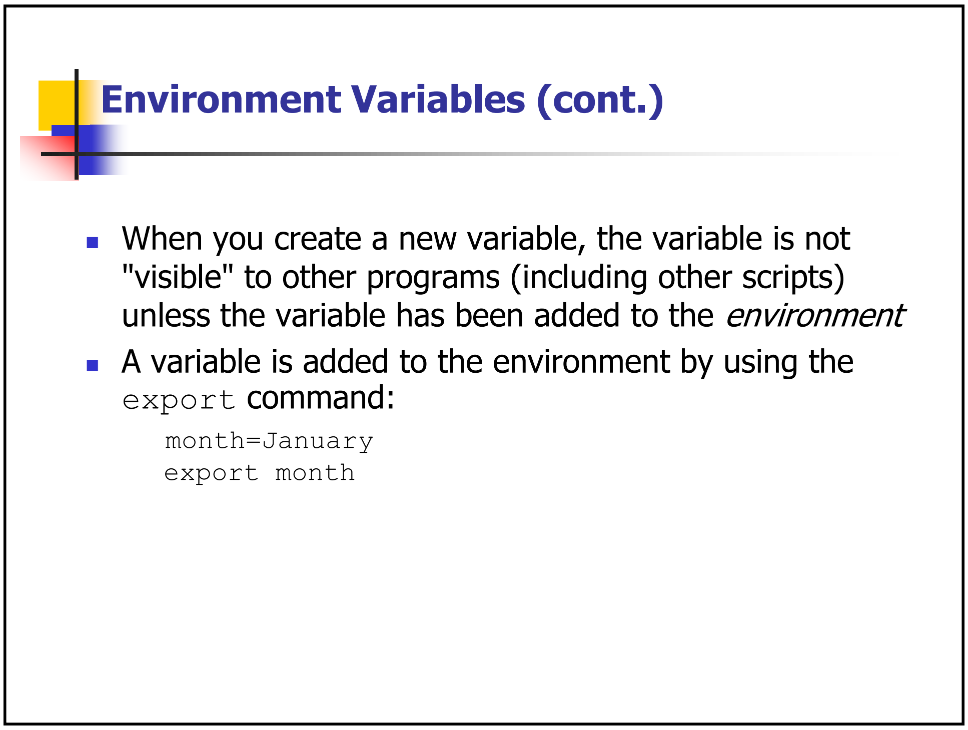
- Many shell variables are "inherited" from the login shell *environment*. In other words, they are preset variables
- For example, when running a script the following variables will be available (amongst others):
  - `HOME`
  - `PATH`
  - `LOGNAME`
  - `TERM`
- Such variables may be changed by the script, but the changes will not be seen by the login shell unless the script was run using the "." operator.

17

Predefined Shell Variables

Shell Variable	Description
PWD	The most recent current working directory.
OLDPWD	The previous working directory.
BASH	The full path name used of the bash shell.
RANDOM	Generates a random integer between 0 and 32,767
HOSTNAME	The current hostname of the system.
PATH	A list of directories to search of commands.
HOME	The home directory of the current user.
PS1	The primary prompt (also PS2, PS3, PS4).

18

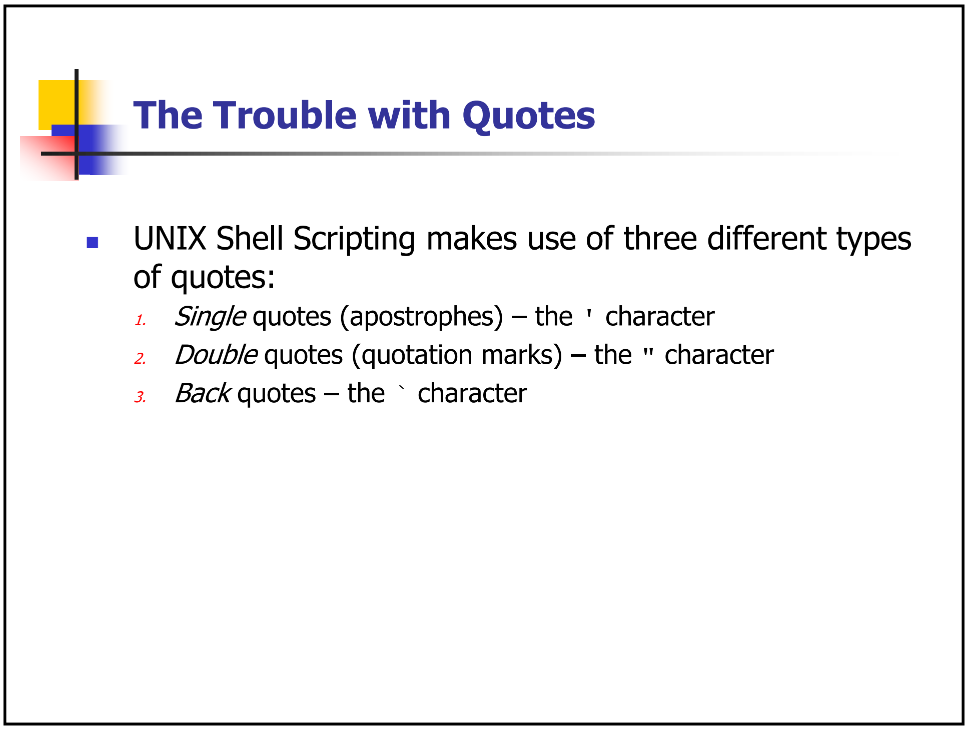


Environment Variables (cont.)

- When you create a new variable, the variable is not "visible" to other programs (including other scripts) unless the variable has been added to the *environment*
- A variable is added to the environment by using the `export` command:

```
month=January
export month
```

19



The Trouble with Quotes

- UNIX Shell Scripting makes use of three different types of quotes:
  1. *Single* quotes (apostrophes) – the `'` character
  2. *Double* quotes (quotation marks) – the `"` character
  3. *Back* quotes – the `` ` `` character

20

Single Quotes

- For example, the shell command:

```
$ echo 'The total is nearly $750'
```

will cause the following output to appear on the screen:

```
The total is nearly $750
```

21

Double Quotes

- Single quotes remove *all* of the shell's special-character features. Sometimes this is excessive – we may prefer *some* of the special characters to work, specifically:
  - `$` (for variable substitution, e.g. `$PATH`)
  - `` ` `` (see the next section)
  - Also, we may want the use of certain constructs, like `\"` or `\$`
- In these situations we can surround the text with *double* quotes. Other characters are still treated as special
- For example:

```
echo "$LOGNAME made \$1000 in `date +%B`"
```

produces

```
peter made $1000 in November
```

22

Back Quotes

- Unlike single and double quotes, the back quotes have nothing to do with special characters
- Any text enclosed in back quotes is treated as a UNIX command, and is executed in its own shell. Any output from the command is substituted into the script line, replacing the quoted text
- For example

```
list=`who | sort`
echo $list
```

produces

```
fred tty02 Aug 21 11:01 peter tty01 Aug 22
09:58 tony tty05 Aug 22 10:32
```

23

Line control

- It is possible to run two or more UNIX commands on the same line in a shell script, by separating the commands with the *;* (semicolon) character
- For example:

```
echo Please enter your Name:; read name
```

- For aesthetic reasons, you may wish to split a command line over more than one line of text. This is achieved by quoting the newline character, using either single quotes, double quotes or the backslash character
- For example:

```
echo This command is split \
over several lines
```

24

Exercises

- Which of the following are valid variable names?
  - A. `month`
  - B. `echo`
  - C. `$year`
  - D. `24_hours`
  - E. `hours-24`
  - F. `fifty%`
  - G. `First Name`
  - H. `a`
  - I. `_First_Name`
  - J. `winner!`

25

Exercise Solutions

- Which of the following are valid variable names?

A. `month`	Valid
B. `echo`	Valid
C. `$year`	The `$` causes the *contents* of the variable to be displayed
D. `24_hours`	Variable names cannot start with a digit
E. `hours-24`	Variable names cannot contain a –
F. `fifty%`	Variable names cannot contain a `%`
G. `First Name`	Variable names cannot contain a space
H. `a`	Valid
I. `_First_Name`	Valid
J. `winner!`	Variable names cannot contain a `!`

26

User-defined Shell Variables

- Syntax:

  `varname=value`

<u>Example:</u>

```
rate=7.65
echo “Rate today is: $rate”
```

- use double quotes if the value of a variable contains white spaces

<u>Example:</u>

```
name=“Thomas William Flowers”
```

27

Use Variables: Work with Variables

- Shell variables

```
tux@da10:~> VARIABLE1="Good morning"
tux@da10:~> echo $VARIABLE1
Good morning
tux@da10:~> bash
tux@da10:~> echo $VARIABLE1

tux@da10:~>
```

- Environment variables

```
tux@da10:~> export VARIABLE2="Good afternoon"
tux@da10:~> bash
tux@da10:~> echo $VARIABLE2
Good afternoon
tux@da10:~>
```

- To see which variables have been set for your shell, use **export**, **set** and **env**

28

Use Variables: Work with Variables (continued)

- Use ***variable=value command*** to execute commands in a modified environment
- Use the command **unset *variable*** to delete a variable

```
tux@da10:~> a=10
tux@da10:~> echo $a
10
tux@da10:~> unset a
tux@da10:~> echo $a

tux@da10:~>
```

29

Conditional Command Execution

- It is possible to specify that a command in a script will only run if particular condition is met
- Such conditions are *always* expressed in terms of the exit status of another program, as follows:

```
command1 && command2
```

means that `command2` will only run if `command1` completes with an exit status of `0`

```
command3 || command4
```

means that `command4` will only run if `command3` completes with an exit status that is *not* `0`

30

Conditional Command Execution

- For example:

```
ls file1 && cp file1 /tmp
cp abc xyz && echo The file was copied okay
diff fileA fileB || echo The files are different
ls file2 || exit
```

- The only problem with these constructs is that they are very limited:
  - You can only perform *one* command if the condition is met (however, it *is* possible to group commands)
  - You cannot specify a second command to be run if the condition is *not* met

31