1.109.2
Customize or write simple scripts
Weight 3

Linux Professional Institute Certification — 102

Nick Urbanik <nicku@nicku.org>
This document Licensed under GPL—see section 18

2005 November
Outline

Context
Objectives
The shebang: #!
Making the script executable
Should you make a script SUID?
True and False
Shell Variables
Special Variables
Quoting
  Quoting and Funny Chars
  Quoting
Command Substitution
The if statement
while statement
The for statement
The test program
  Conditions
Arithmetic
Input & Output
  Output with echo
  Input with read
Alerting about problems by email
License Of This Document
Topic 109 Shells, Scripting, Programming and Compiling [8]
Where we are up to

1.109.1 Customize and use the shell environment [5]
1.109.2 Customize or write simple scripts [3]
Candidate should be able to customize existing scripts, or write simple new (ba)sh scripts. This objective includes using standard sh syntax (*loops, tests*), using *command substitution*, testing *command return values*, testing of *file status*, and conditional *mailing to the superuser*. This objective also includes making sure the correct interpreter is called on the first (*#!*) line of scripts. This objective also includes managing *location, ownership*, execution and *suid-rights* of scripts.
Key files, terms, and utilities include:

1.109.2 Customize or write simple scripts [3]

while — shell builtin: does things repetively while a condition is true
for — shell builtin: does things repetitively, once with each element of a list
test — used to construct a condition
chmod — an external command, to change the permission on a file
The Shebang: #!

- You ask the Linux kernel to execute the shell script
- kernel reads first two characters of the executable file
  - If first 2 chars are “#!” then
  - kernel executes the name that follows, with the file name of the script as a parameter
- Example: a file called `find.sh` has this as the first line:
  ```sh
  #! /bin/sh
  ```
- then kernel executes this:
  ```sh
  /bin/sh find.sh
  ```
- What will happen in each case if an executable file begins with:
  ```sh
  #! /bin/rm
  #! /bin/ls
  ```
Making the script executable

To easily execute a script, it should:

- be on the **PATH**
- have execute permission.

How to do each of these?

- Red Hat Linux by default, includes the directory `~/bin` on the **PATH**, so create this directory, and put your scripts there:
  
  ```bash
  $ mkdir ~/bin →
  
  $ chmod +x script →
  ```
Should you make a script SUID?

- Normally, when you run a script, the process is owned by you, and has the same access rights as you.
- If a script has the SUID permission, then:
  - it does not matter who executes it!
  - the owner of the process is the owner of the file
  - This is very dangerous, especially if the owner of the file is root!
- Never make a shell script SUID, unless you really, really know what the risks are and how to avoid them.
- Instead, write it in a language such as Perl, with taint checking, and make it as simple as possible.
- See Topic 1.114.1 Perform security administration tasks for details of manipulating SUID/SGID permissions.
Shell programs depend on executing external programs

When any external program execution is successful, the exit status is zero, 0

An error results in a non-zero error code

To match this, in shell programming:
  - The value 0 is true
  - any non-zero value is false

This is opposite from other programming languages
Variables—1

- Variables not declared; they just appear when assigned to
- Assignment:
  - no dollar sign
  - no space around equals sign
  - examples:
    - `$ x=10` # correct
    - `$ x = 10` # wrong: try to execute program called “x”
- Read value of variable:
  - put a ‘$’ in front of variable name
  - example:
    - `$ echo "The value of x is $x"`
You can put **multiple assignments** on one line:
```
i=0  j=10  k=100
```

You can **set a variable temporarily** while executing a program:
```
$ echo $EDITOR
emacsclient
$ EDITOR=gedit crontab -e
$ echo $EDITOR
emacsclient
```
Variables—Local to Script

- Variables disappear after a script finishes
- Variables created in a sub shell disappear
  - *parent shell cannot read variables in a sub shell*
  - example:
    ```
    $ cat variables
    #! /bin/sh
    echo $HOME
    HOME=happy
    echo $HOME
    $ ./variables
    /home/nicku
    happy
    $ echo $HOME
    /home/nicku
    ```
You can make a variable hold the null string by assigning it to nothing, but it does not disappear totally:

```bash
$ VAR=
$ env | grep '^VAR'
VAR=
```

You can make it disappear totally using `unset`:

```bash
$ unset VAR
$ env | grep '^VAR'
```
Command-line Parameters

- Command-line parameters are called $0, $1, $2, ...

- Example: when call a shell script called “shell-script” like this:

  
  ```
  $ shell-script param1 param2 param3 param4
  ```

<table>
<thead>
<tr>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>shell-script</td>
</tr>
<tr>
<td>$1</td>
<td>param1</td>
</tr>
<tr>
<td>$2</td>
<td>param2</td>
</tr>
<tr>
<td>$3</td>
<td>param3</td>
</tr>
<tr>
<td>$4</td>
<td>param4</td>
</tr>
<tr>
<td>$#</td>
<td>number of parameters to the program, e.g., 4</td>
</tr>
</tbody>
</table>

- Note: these variables are read-only.
Special Built-in Variables

- Both $@ and $* are a list of all the parameters.
- The only difference between them is when they are quoted in quotes—see manual page for `bash`
- $? is exit status of last command
- $$ is the process ID of the current shell
- Example shell script:

```bash
#!/bin/sh
echo $0 is the full name of this shell script
echo first parameter is $1
echo first parameter is $2
echo first parameter is $3
echo total number of parameters is $#
echo process ID is $$
```
## Special Characters

Many characters have a special meaning to the shell.

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Home directory</td>
</tr>
<tr>
<td>`</td>
<td>Command substitution. Better: $(...)</td>
</tr>
<tr>
<td>#</td>
<td>Comment</td>
</tr>
<tr>
<td>$</td>
<td>Variable expression</td>
</tr>
<tr>
<td>&amp;</td>
<td>Background Job</td>
</tr>
<tr>
<td>*</td>
<td>File name matching wildcard</td>
</tr>
<tr>
<td></td>
<td>Pipe</td>
</tr>
<tr>
<td>(</td>
<td>Start subshell</td>
</tr>
<tr>
<td>)</td>
<td>End subshell</td>
</tr>
<tr>
<td>[</td>
<td>Start character set file name matching</td>
</tr>
<tr>
<td>]</td>
<td>End character set file name matching</td>
</tr>
<tr>
<td>{</td>
<td>Start command block</td>
</tr>
<tr>
<td>;</td>
<td>Command separator</td>
</tr>
<tr>
<td>\</td>
<td>Quote next character</td>
</tr>
<tr>
<td>'</td>
<td>Strong quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>Weak quote</td>
</tr>
<tr>
<td>&lt;</td>
<td>Redirect Input</td>
</tr>
<tr>
<td>&gt;</td>
<td>Redirect Output</td>
</tr>
<tr>
<td>/</td>
<td>Pathname directory separator</td>
</tr>
<tr>
<td>?</td>
<td>Single-character match in filenames</td>
</tr>
<tr>
<td>!</td>
<td>Pipeline logical NOT</td>
</tr>
</tbody>
</table>

**⟨space or tab⟩** shell normally splits at white space
Quoting

- Sometimes you want to use a special character \textit{literally}; i.e., without its special meaning.
- Called \textit{quoting}
- Suppose you want to print the string: $2 \times 3 > 5$ is a valid inequality?
- If you did this:
  
  $\$ \texttt{echo 2 \times 3 > 5 is a valid inequality}$

  the new file ‘5’ is created, containing the character ‘2’, then the names of all the files in the current directory, then the string “3 is a valid inequality”.

License Of This Document

The if statement

while statement

The for statement

The test program

Arithmetic

Input & Output

Alerting about problems by email

License Of This Document
To make it work, you need to protect the special characters ‘*’ and ‘>’ from the shell by quoting them. There are three methods of quoting:

- Using double quotes (“weak quotes”)
- Using single quotes (“strong quotes”)
- Using a backslash in front of each special character you want to quote

This example shows all three:

```bash
$ echo "2 * 3 > 5 is a valid inequality"
$ echo '2 * 3 > 5 is a valid inequality'
$ echo 2 \* 3 \> 5 is a valid inequality
```
Quoting—When to use it?

- Use quoting when you want to pass special characters to another program.
- Examples of programs that often use special characters:
  - find, locate, grep, expr, sed and echo
- Here are examples where quoting is required for the program to work properly:
  ```bash
  $ find . -name \*.jpg
  $ locate '/usr/bin/c*'
  $ grep 'main.*/(.c
  $ i=$(expr i \* 5)
  ```
More about Quoting

- **Double quotes**: "..." stop the special behaviour of all special characters, except for:
  - variable interpretation ($)
  - backticks (‘) — see slide 5
  - the backslash (\)

- **Single quotes**: ’...’:
  - stop the special behaviour of *all* special characters

- **Backslash**:
  - preserves literal behaviour of character, except for newline; see slides §4, §9
  - Putting “\” at the end of the line lets you continue a long line on more than one physical line, but the shell will treat it as if it were all on one line.
Command Substitution — $(...) or `...`

- Enclose command in $(...) or backticks: `...`
- Means, “Execute the command in the $(...) and put the output back here.”
- Here is an example using expr:
  ```
  $ expr 3 + 2
  5
  
  $ i=expr 3 + 2  # error: try execute command ‘3’
  $ i=$(expr 3 + 2)  # correct
  $ i=`expr 3 + 2`  # also correct
  ```
Command Substitution—Example

- We want to put the **output of the command** `hostname` into a **variable**:
  
  $$\text{hostname}$$
  
  $$\text{nicku.org}$$
  
  $$\text{h}=$$ `hostname`
  
  $$\text{echo } \$h$$
  
  `hostname`
  
- Oh dear, we only stored the **name** of the command, not the **output** of the command!
  
- **Command substitution** solves the problem:
  
  $$\text{h}=$(\text{hostname})$$
  
  $$\text{echo } \$h$$
  
  `nicku.org`
  
- We put `$(...)` around the command. You can then assign the output of the command.
if Statement

▶ Syntax:

```bash
if ⟨test-commands⟩
then
  ⟨statements-if-test-commands-1-true⟩
elif ⟨test-commands-2⟩
then
  ⟨statements-if-test-commands-2-true⟩
else
  ⟨statements-if-all-test-commands-false⟩
fi
```

▶ Example:

```bash
if grep nick /etc/passwd > /dev/null 2>&1
then
  echo Nick has a local account here
else
  echo Nick has no local account here
fi
while Statement

- Syntax:

```bash
while ⟨test-commands⟩
do
  ⟨loop-body-statements⟩
done
```

- Example:

```bash
i=0
while [ "$i" -lt 10 ]
do
  echo -n "$i " # -n suppresses newline.
  let "i = i + 1" # i=$(expr $i + 1) also works
done
```
for Statement

- Syntax:

```bash
for ⟨name⟩ in ⟨words⟩
do
  ⟨loop-body-statements⟩
done
```

- Example:

```bash
for planet in Mercury Venus Earth Mars \ 
  Jupiter Saturn Uranus Neptune Pluto
do
  echo $planet
done
```

- The backslash "\" quotes the newline. It's just a way of folding a long line in a shell script over two or more lines.
for Loops: Another Example

- Here the shell turns `*.txt` into a list of file names ending in "*.txt":

```bash
for i in *.txt
do
    echo $i
grep 'lost treasure' $i
done
```

- You can leave the `in ⟨words⟩` out; in that case, ⟨name⟩ is set to each parameter in turn:

```bash
i=0
for parameter
do
    let 'i = i + 1'
    echo "parameter $i is $parameter"
done
```
Conditions—String Comparisons

- All programming languages depend on *conditions* for *if* statements and for *while* loops.

- Shell programming uses a built-in command which is either *test* or [*...*]

- Examples of *string* comparisons:

  ```
  [ "$USER" = root ]  # true if the value of $USER is "root"
  [ "$USER" != root ]  # true if the value of $USER is not "root"
  [ -z "$USER" ]       # true if the string "$USER" has zero length
  [ string1 < string2 ] # true if string1 sorts less than string2
  [ string1 > string2 ] # true if string1 sorts greater than string2
  ```

- Note that we need to quote the ‘>’ and the ‘<’ to avoid interpreting them as file redirection.

- *Note*: the spaces after the “[“ and before the “]” are essential.

- Also spaces are *essential* around operators.
Examples of *numeric* integer comparisons:

```
[ "$x" -eq 5 ]  # true if the value of $x is 5
[ "$x" -ne 5 ]  # true if integer $x is not 5
[ "$x" -lt 5 ]  # true if integer $x is < 5
[ "$x" -gt 5 ]  # true if integer $x is > 5
[ "$x" -le 5 ]  # true if integer $x is ≤ 5
[ "$x" -ge 5 ]  # true if integer $x is ≥ 5
```

Note again that the spaces after the "[“ and before the “]” are essential.

Also spaces are *essential* around operators.
Conditions—File Tests, NOT Operator

- The shell provides many tests of information about files.
- Do `man test` to see the complete list.
- Some examples:

```bash
$ [ -f file ]  # true if file is an ordinary file
$ [ ! -f file ]  # true if file is NOT an ordinary file
$ [ -d file ]  # true if file is a directory
$ [ -u file ]  # true if file has SUID permission
$ [ -g file ]  # true if file has SGID permission
$ [ -x file ]  # true if file exists and is executable
$ [ -r file ]  # true if file exists and is readable
$ [ -w file ]  # true if file exists and is writeable
$ [ file1 -nt file2 ]  # true if file1 is newer than file2
```

- **Note again:** the spaces after the “[“ and before the “]” are essential.
- Also spaces are essential around operators.
Examples of combining comparisons with AND: −a and OR: −o, and grouping with \ ( . . . \)

# true if the value of $x$ is 5 AND $USER$ is not equal to root:
[ "$x" -eq 5 -a "$USER" != root ]

# true if the value of $x$ is 5 OR $USER$ is not equal to root:
[ "$x" -eq 5 -o "$USER" != root ]

# true if ( the value of $x$ is 5 OR $USER$ is not equal to root ) AND
# ( $y > 7 OR $HOME has the value happy )
[ \( "$x" -eq 5 -o "$USER" != root \) -a \( "$y" -gt 7 -o "$HOME" = happy \) ]

Note again that the spaces after the “[“ and before the “]” are essential.

Do man test to see the information about all the operators.
Arithmetic Assignments

- Can do with the external program `expr`
  - ... but `expr` is not so easy to use, although it is very standard and *portable*: see `man expr`
  - Easier is to use the built in `let` command
    - see `help let`

- Examples:
  ```
  $ let x=1+4
  $ let ++x  # Now x is 6
  $ let x='1 + 4'
  $ let 'x = 1 + 4'
  $ let x="(2 + 3) * 5"  # now x is 25
  $ let "x = 2 + 3 * 5"  # now x is 17
  $ let "x += 5"        # now x is 22
  $ let "x = x + 5"     # now x is 27; NOTE NO $
  ```

- Notice that you do not need to quote the special characters with `let`.
- Quote if you want to use white space.
- Do not put a dollar in front of variable, even on right side of assignment; see last example.
Output with `echo`

- To perform output, use `echo`, or for more formatting, `printf`.
- Use `echo -n` to print no newline at end.
- Just `echo` by itself prints a newline.
Input: the `read` Command

- For input, use the built-in shell command `read`
- `read` reads standard input and puts the result into one or more variables
- If use one variable, variable holds the whole line
- Syntax:
  
  ```bash
  read ⟨var1⟩... 
  ```
- Often used with a `while` loop like this:
  ```bash
  while read var1 var2 
  do 
      # do something with $var1 and $var2 
  done
  ```
- Loop terminates when reach end of file
- To prompt and read a value from a user, you could do:
  ```bash
  while [ -z "$value" ]; do 
      echo -n "Enter a value: " 
      read value 
  done 
  ```
  # Now do something with $value
Your Linux system has a large number of shell scripts that you can refer to as examples. I counted about 1400. Here is one way of listing their file names:

```
$ file /bin/* /usr/bin/* /usr/sbin/* /sbin/* /etc/rc.d/* /usr/X11R6/bin/* | grep -i "shell script" | awk -F: '{print $1}''
```

Let's see how this works. I suggest executing the commands separately to see what they do:

```
$ file /bin/* /usr/bin/*
$ file /bin/* /usr/bin/* | grep -i "shell script"
$ file /bin/* /usr/bin/* | grep -i "shell script" | awk -F: '{print $1}''
```

The **awk** program is actually a complete programming language. It is mainly useful for selecting columns of data from text. **awk** automatically loops through the input, and divides the input lines into fields. It calls these fields $1, $2, ..., $NF. $0 contains the whole line. Here the option -F: sets the **field separator** to the colon character. Normally it is any white space. So printing $1 here prints what comes before the colon, which is the file name.

Suppose you want to look for all shell scripts containing a particular command or statement? Looking for example shell scripts that use the `mktemp` command:
Customize or write simple scripts

Weight 3

Nick Urbanik

Context
Objectives

The shebang: `#!`

Making the script executable

Should you make a script SUID?

True and False

Shell Variables

Special Variables

Quoting

Command Substitution

The if statement

while statement

The for statement

The `test` program

Arithmetic

Input & Output

Alerting about problems by email

License Of This Document
Alerting about problems by email

#!/bin/sh

# A quick script whipped up by Nick to send mail if
# root file system is more than 90 per cent full.

percentful=$(df / | awk 'NR > 1{sub("%", ",", $5);print $5}')

if [ "$percentful" -gt 90 ]
then
    message="root file system is $percentful% full"
    echo "$message" | mail -s $message root
fi
Topics Covered

Context
Objectives
The shebang: #!
Making the script executable
Should you make a script SUID?
True and False
Shell Variables
Special Variables
Quoting
  Quoting and Funny Chars
Quoting

Command Substitution
The if statement
while statement
The for statement
The test program
  Conditions
Arithmetic
Input & Output
  Output with echo
  Input with read
Alerting about problems by email
License Of This Document
License Of This Document

Copyright © 2005 Nick Urbanik <nicku@nicku.org>
You can redistribute modified or unmodified copies of this
document provided that this copyright notice and this
permission notice are preserved on all copies under the terms
of the GNU General Public License as published by the Free
Software Foundation — either version 2 of the License or (at
your option) any later version.