<<< Previous Next >>>

Basic Shell Scripting Using bash

When working in a terminal window, the shell is the program that reads input that you type in, forwards it to the operating system and then presents a prompt accepting the next input. It is therefore your interface to directly launching commands. Its power comes from the way how commands can be combined and wrapped up in scripts to perform more complex or repetitive operations. Using shell commands efficiently can save you a lot of time, avoiding repetition be it on the command line or in a graphical interface.

Slides

Slides accompanying this part of the introduction cover basic shell scripting and control structures.

Hello world

The bash version of the ubiquitous hello world example is below. The line starting with **#!** specifies the shell, i.e. the program to execute this file. The **echo** command prints the following string. The string can contain variables, which will be expanded, i.e. replaced by their contents.

Example 1. Hello World

```
#!/bin/bash
echo "Hello world"
```

Cut-and-paste this code into a file called `hello.sh`. Then set execute permission on that file and run it like this:

```
$ chmod a+x hello.sh
$ ./hello.sh
Hello World
```

Shell variables

Example 2. Hello User

```
#!/bin/bash
echo "Hello $USER"
```

Bash distinguishes between read and write usages of shell variable. In a read usage, denoted by a **$** prefix, the occurence of the variable is replaced by its value. In a write usage, just using the name of the variable, the variable stands for the location where to save a value. The following example first makes write use of variable **v**, storing a string, and then makes a read use, expanding and printing the contents of the variable. The variable **USER** is a so-called *environment variable*, which is automatically set when logging in.

Example 3. Hello User

```
#!/bin/bash
v="Hello $USER"
echo $v
```

Changing your prompt

The following command modifies your prompt to display username, machinename, history index and the version of the bash you are using. It is an example how to assign to an environment variable.

```
export PS1="\u@\h[\!](\v)> "
```

Expansion of variables under strings is done differently, depending on the style of quotes that are used:

- Within double-quotes (**"**), variables will be expanded and a sub-shell can be executed using back-ticks **"**
- Within single-quotes (**'**), all characters are taken literally and no variable expansion is executed
- If variable expansion is enabled, special characters can be escaped using a backslash (\)

```
$ what="variable expansion"
$ s="testing $what"
$ echo $s
testing variable expansion
$ s="testing \$what"
$ echo $s
testing $what
$ s='testing $what'
$ echo $s
testing $what
t="The current date is `date`"
```

```
$ echo $t
The current date is Thu Sep 6 16:26:42 BST 2012
```

Shell Quoting

Expansion of shell variables can be tricky. For more details see the Advanced Shell Scripting Guide.

Other special variables are

- **$#** the number of arguments.
- **$*** the entire argument string.
- **$?** the return code of the last command issued.

Arithmetic expressions

In a previous example we used the following pipe to compute the number of files in the current directory:

```
$ ls -l | wc -l
```

We noted, that this is off by one, since ls also reports a total. We can account for that by performing arithmetic on the result. We first need to assign the result to a shell variable. Then we use a let binding, which performs arithmetic:

```
$ x=`ls -l | wc -l`
52
$ let n=x-1
$ echo $n
51
```

Or in short (the $[ ] notation separates arithmetic evaluation from bindings):

```
$ echo $[ `ls -l | wc -l` - 1]
51
```

See this section in the Advanced Bash-Scripting Guide for details on arithmetic expressions, backtics and let-bindings.

Examples of Regular Expressions

The following annotated sequence of commands demonstrates the usage of regular expressions, by selecting files and searching for text in files.

```
# list all C source or header files
$ ls *.[ch]
# search for Flags in these files
$ egrep Flags *.[ch]
# colourise
$ egrep -n --color Flags *.[ch]
# everything that accesses Flags, ie. a '.' afterwards
$ egrep -n --color "(Par|Gc)Flags[.]" *.[ch]
# several dots
$ egrep -n --color "(Par|Gc)Flags[.]{3,}" *.[ch]
# Flags but not with an 'n' after the '.'
$ egrep -n --color "(Par|Gc)Flags[.][^n]" *.[ch]
# recursively in all dirs
$ find . -name "*.[ch]" -print
# do something recursively in all dirs
$ find . -name "*.[ch]" -print -exec egrep -n --color "(Par|Gc)Flags[.][^n]" \{\} \;
# files modified within the last 24 hrs
$ find . -mtime -1 -print

# disk usage
$ du -s *.[ch] | sort -n | (while read x nam ; do z=$[ $z + $x ] ; done; echo $z)


# Pipes etc

# check CVS status of the dir
$ cvs status .
# narrow it down to the main status line
$ cvs status . | egrep "^File"
# ignore all up-to-date lines
$ cvs status . | egrep "^File" | sed '/Up-to-date/d'
# ignore the "examning" output
$ cvs status . 2>/dev/null | egrep "^File" | sed '/Up-to-date/d'

# Getting a list of all working revision numbers
# extract the revision number, using sed
$ cvs status . | sed 's/Working revision[^0-9]*\([.0-9]*\).*$/\1/'
# ignore all other lines
$ cvs status . | sed '/Working revision/!d;s/Working revision[^0-9]*\([.0-9]*\).*$/\1/'
# sort the numbers
$ cvs status . | sed '/Working revision/!d;s/Working revision[^0-9]*\([.0-9]*\).*$/\1/' | sort
# delete duplicate lines
$ cvs status . | sed '/Working revision/!d;s/Working revision[^0-9]*\([.0-9]*\).*$/\1/' | sort | uniq
# ignore stdout
$ cvs status . 2>/dev/null | sed '/Working revision/!d;s/Working revision[^0-9]*\([.0-9].*\).*$/\1/' | sort | uniq
```

```
$ loc1.sh *.[ch]
```

The following example takes a log file of a series of parallel runs and turns it into a gnuplot-generated speedup graph. Details on which data is shown is not important. It mainly serves as an example on how to use basic Linux tools, composed via pipes, to do data processing (with the alias **qp** defined below) and visualisation (using the GNU tool **gnuplot**) from the command-line or from within a shell script. After running the last command you should see a new window with a speedup graph.

```
# Input: sequence of ghc -sstderr outputs, first one is sequential runtime
# Output: .gp files of speedups
# NB: this must be executed in the .bashrc; on the commandline more escaping is needed
$ alias qp="cat LOG | sed -n '/SPARKS/p;/^\*/p;/Total time/p' | sed '/Total time/a\X' | sed '/SPARKS/d;s/^.*PEs \([0-9]*\)/\1/;s/^.*Total

# download the data file, containing measurement data, via the web
$ wget http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/Data/LOG-2010-06-10-0105
# create symbolic link, ie. an indirection, from LOG-2010-06-10-0105 to LOG; both names now refer to the same file
$ ln -s LOG-2010-06-10-0105 LOG
# now calculate the speedups for all executions in the data file:
#   ignore everything after Strategy 3, then delete comment lines, and take last 8 lines
$ qp | sed '/Strategy 2/q' | sed '/^#/d' | tail -8 >q.dat
# showing speedup the speedup graph
$ echo "set term x11; plot \"q.dat\" with lines; pause 10 " | gnuplot
```

An Overview of Bash Control Structures

Similar in behaviour to the if statement in other programming languages with few differences: The command is executed which could be a script. Based on the return value, the statement after the then or else get executed. Return value 0 is true and any other value is considered to be false.

Online examples

Some examples in this section are taken from the Linux Classes online course.

```
if script1; then
  ls -l
else
  echo "script1 returned false"
fi
```

Example 4. Conditionals

```
#!/bin/bash
FIRST_ARGUMENT="$1"
if [ x$FIRST_ARGUMENT = "xHans" ]; then
  echo "Hello Hans, good to see you"
else
  echo "Hello World"
fi
```

Example 5. File Copy

```
#!/bin/bash
# copy a file, creating a backup if the target file exists
if [ $# -lt 2 ]
then
 echo "Usage: $0 fromfile tofile"
 exit 1
fi
if [ -f $2 ]
 then mv $2 $2.bak
fi
cp $1 $2
```

Loops

Example 6. While loop

```
#!/bin/bash
while [ "$*" != "" ]
do
  echo "value is $1"
  shift
done
```

This script prints all arguments passed to it, using a while loop. The **shift** command moves the values stored in the command line variables one position to the left. The first command line argument is **$1**, and a **$*** is a string composed of all arguments.

```
$ ./echoall.sh one two three
value is one
value is two
value is three
```

The following example uses an until loop instead:

Example 7. Until loop

```
#!/bin/bash
# Source: http://lowfatlinux.com/linux-script-looping.html

count=1
until [ "$*" = "" ]
do
  echo "value number $count $1 "
  shift
  count=$[ $count + 1 ]
  # count= `expr $count + 1`
done
```

The following example uses a for loop instead:

Example 8. For loop

```
#!/bin/bash
# Source: http://lowfatlinux.com/linux-script-looping.html
for item in "$*"
do
  echo "value is $item"
done
```

Shell Functions

Functions in shell-scripts are very restricted, and the syntax is given below. The () indicates that this is a function, but it is *always empty*. Arguments are referred to in the body of the function as `$1, $2` etc. A function may return an byte-size integer value as the return code.

```
functionname () {
 statement
}
```

See the section on functions in the Advanced Bash-Scripting Guide:.

Further Reading

Study some of the examples in the Advanced Bash Scripting Guide, for example this days-between function.

Exercise

As a simple exercise, write a shell script that prints the file type for each file passed as an argument to the script, e.g.

```
$ ./ftys.sh ftys.sh .emacs .bashrc public_html
ftys.sh: Bourne-Again shell script text executable
.emacs: Lisp/Scheme program text
.bashrc: ASCII English text, with very long lines
public_html: symbolic link to `www'
```

Hint: Check the Section called *Other useful commands* for some useful commands.