Bash Scripting Part 4

OPS102 Week 9 Class 2

Chris Tyler/John Sellens

July 11, 2024

Seneca Polytechnic

Outline

Recap From Last Class

Recap From Last Class

- The `"test"` command is the best.
- Now, what more can we do?

Script Parameters

Script Parameters

- It's useful to be able to call a script with positional parameters (arguments).
- These can be accessed within a script as $0, $1, $2, $3, and so forth.
- $0 is the name of the script itself.
- $# is the number of positional parameters.
- The shift command gets rid of the first parameter ($1) and shifts every parameter to a lower number.

Script Parameters

```
$ cat params
#!/usr/bin/bash
echo "Number of parameters:  $#"
echo "Parameter 1:           $1"
echo "Parameter 2:           $2"
echo "Parameter 3:           $3"
echo "Parameter 4:           $4"

$ ./params red green blue
Number of parameters:  3
Parameter 1:           red
Parameter 2:           green
Parameter 3:           blue
Parameter 4:
```

Script Parameters and shift

```
$ cat params
#!/usr/bin/bash
echo "Number of parameters:  $#"
shift
echo "Parameter 1:           $1"
echo "Parameter 2:           $2"
echo "Parameter 3:           $3"
echo "Parameter 4:           $4"
$ ./params red green blue
Number of parameters:  3
Parameter 1:           green
Parameter 2:           blue
Parameter 3:
Parameter 4:
```

Script Parameters in Totality

- `$*` and `$@` both return ALL of the parameters.
- When quoting:
  - `"$*"` returns all the parameters as a single string –- not usually useful.
  - `"$@"` returns each parameter as a separate string –- usually what you want.

Script Parameters in Totality Example

```
$ cat params3
#!/usr/bin/bash
ls –l "$*"     # gives one file argument to ls
echo ---
ls –l "$@"     # gives separate file arguments to ls
$ touch a b c
$ ./params3 a b c
ls: cannot access 'a b c': No such file or directory
---
-rwxr-xr-x. 1 chris chris 463 Jun 21 11:55 a
-rwxr-xr-x. 1 chris chris 121 Jun 21 11:55 b
-rwxr-xr-x. 1 chris chris 532 Jun 21 11:55 c
```

Script Parameters Another Example

- Let's look at a simple bash script to check that the user has provided 2 arguments.
- In this script, we're also including the name of the script in the error message, sending the error message to stderr, and exiting with a unique error code.

Script Parameters Another Example

```
$ cat paramcheck
#!/usr/bin/bash
if [[ "$#" -ne 2 ]]
then
  echo "$(basename $0): Error: 2 arguments expected" >&2
  exit 1
fi
exit 0

$ ./paramcheck foo bar
$ ./paramcheck foo
paramcheck: Error: 2 arguments expected
```

Looping in Bash Scripts

Looping in Bash Scripts

There are four types of loops available in bash:

```
for variable in values ; do ... ; done
```

```
for (( setup; control; iteration )) ; do ... ; done
```

```
while cmdlist ; do ... ; done
```

```
until cmdlist ; do ... ; done
```

Looping: for *variable* in *values*

- This type of loop accepts a list of values. The first value is assigned to the variable and the loop is executed, and then the process is repeated with each remaining value.
- The *values* could be:
  - A list of constants:

    ```
    for CITY in Toronto Vaughan Oshawa ; do ... ; done
    ```

  - Parameters:

    ```
    for X in "$@" ; do ... ; done
    ```

  - A file globbing pattern:

    ```
    for FILE in *.jpg ; do ... ; done
    ```

- Or anything else that consists of one or more values as separate words.
- Or even an empty list of values.

Looping: for *variable* in *values*

```
$ cat tidyup
#!/usr/bin/bash
for FILE in *.backup *.bck ; do
  if [[ -r "$FILE" ]] ; then
    read -p "Delete file '$FILE' (Y/N)? " YESNO
    if [[ "$YESNO" == "y" || "$YESNO" == "Y" ]] ; then
      echo "Deleting file '$FILE'"
      rm "$FILE"
    else
      echo "'$FILE' was not deleted."
    fi
  fi
done
```

Looping: for *variable* in *values*

```
$ touch oldfile.backup source.bck
$ ./tidyup
Delete file 'oldfile.backup' (Y/N)? N
'oldfile.backup' was not deleted.
Delete file 'source.bck' (Y/N)? Y
Deleting file 'source.bck'
```

Looping: for ((*setup*; *control*; *iteration*))

- This type of loop works pretty much the same as a C-style for loop.
- Example:

```
for (( i=0; i<10; i++ ))
do
  echo "$i"
done
```

- Remember the double-parenthesis! As with arithmetic!
  - It's "arithmetic context" inside the (( ))

Looping: while *cmdlist*; do . . . ; done

- This type of loop executes as long as the cmdlist returns success
  - i.e. while exitstatus == 0
- Example:

```
while [[ "$(who | wc –l)" -gt 1 ]]
do
  echo "There are other users logged in:"
  who
  sleep 10
done
```

Looping: until *cmdlist*; do . . . ; done

- This type of loop executes as long as the cmdlist does *not* return success
  - i.e. while exitstatus != 0
- Example:

```
until [[ "$(date +%u)" == "6" ]]
do
  echo "Waiting until Saturday..."
  sleep $((24 * 60 * 60))  # sleep for a day
done
```

- This is effectively “while not”

Using Scripts, and Startup Files

Using Scripts

- Scripts are handy for repetitive or complicated tasks.
- Scripts may also be used to customize your environment on a Linux system.

Bash Startup Scripts

- There are two scripts in your home directory that are executed automatically by `bash`. They are both named starting with a period (dot), which causes them to be “hidden” (not normally displayed by the ls command).
- `~/.bash_profile` -– this script is executed once per login.
  - This is a good place to put commands that set up your work environment, including envars, and source your `.bashrc` file.
- `~/.bashrc` -– this script is executed whenever a bash process starts (which may be several times per login session).
  - This is the right place to put things such as command aliases (which are not inherited by child processes).

Startup Scripts – Be Careful!

- A broken ~/.bash_profile or ~/.bashrc script may **prevent** you from successfully logging in to your account!
- To protect yourself:
  - . Test ~/.bash_profile and ~/.bashrc scripts while logged in to your account by explicitly specifying their names. e.g.

    ```
    $ bash ~/.bash_profile
    ```

  - If that is successful, stay logged in to your account while initiating a new login to test the scripts in the login context. For example, if you are logging in remotely, *stay logged in* on one `ssh` session while initiating a new, separate `ssh` login session to test the scripts.

Summary

Summary

- Script parameters and looping constructs? More powerful programs!
- Startup sripts let you customize your working environment.
- Scripting is fun!
  - For certain values of “fun”.