Birzeit University - Faculty of Engineering & Technology
Electrical & Computer Engineering Department - ENCS313
Linux laboratory

Experiment #6
Shell scripting
Part III

0.1 Introduction

The experiment intends to present to students the shell scripting. Students will be shown how to take advantage of the Linux commands they have seen so far in order to build shell scripts. Students will be shown as well how to create and use loops (`for`, `while`, `until`), will be shown the `continue` and `break` constructs as well as the usage of options in shell scripts (`getopts` command). The `printf` command will be presented and students will see the power of formatting the output. Afterwards, students will be shown on how to request user input through the `read` command[1].

The above-mentioned topics are too large to be covered in a single experiment. This is why three experiments will be dedicated to tackle shell scripting.

0.2 Objectives

The objectives of the experiment is to learn the following:

- Provide more insight into the syntax of shell scripting.
- Provide examples on how to use loops in shell scripts.
- Show students how to request user input through the `read` command.

0.3 The `for` command

The `for` command is used to execute a set of commands a specified number of times. Its basic format is as shown:

```
for var in word1 word2 ... wordn
do
  command
  command
  ...
done
```

The commands enclosed between the `do` and the `done` form what's known as the body of the loop. These commands are executed for as many words as you have listed after the `in`. When the loop is executed, the first word, `word1`, is assigned to the variable `var`, and the body of the loop is then executed. Next, the second word in the list, `word2`, is assigned to `var`, and the body of the loop is executed. This process continues with successive words in the list being assigned to `var` and the commands in the loop body being executed until the last word in the list, `wordn`, is assigned to `var` and the body of the loop executed.

Example 1

Create a file called `for_1` and type the following:

[1]Stephen G. Kochan, Patrick Wood - Unix Shell Programming, 3rd Edition

```
#!/bin/sh

for i in 1 2 3
do
  echo $i
done
```

Make the script executable and run it as follows:

`./for_1`

Note the output that you get.

Example 2

Create a file called `for_2` and type the following:

```
#!/bin/sh

echo Number of arguments passed is $#

for arg in $*
do
  echo $arg
done
```

Make the script executable and run it as follows:

`./for_2 my name is coco`

and note the output that you get.

Example 3

Create a file called `for_3` and type the following:

```
#!/bin/sh

for i in `\ls`
do
  echo $i | tr '[a-z]' '[A-Z]'
done
```

Make the script executable and run it as follows:

`./for_3`

Note the output that you get.

0.4 The `for` Without the List

If the `for` command is not followed by the `in` construct as shown above, the shell automatically sequences through all the arguments typed on the command line.

Example 4

Create a file called `for_4` and type the following:

```
#!/bin/sh

echo Number of arguments passed is $#

for arg
do
```

```
  echo $arg
done
```

Make the script executable and run it as follows:

```
./for_4 a b c
```

Note the output that you get.

0.5 The `while` Command

This is the second form of looping in shell scripts. The format is as follows:

```
while commandt
do
  command
  command
  ...
done
```

In the above format, `commandt` is executed and its exit status tested. If it's zero, the commands enclosed between the `do` and `done` are executed. Then `commandt` is executed again and its exit status tested. If it's zero, the commands enclosed between the `do` and `done` are once again executed. This process continues until `commandt` returns a nonzero exit status. At that point, execution of the loop is terminated. Execution then proceeds with the command that follows the `done`.

Example 1

Create a file called `while_1` and type the following:

```
#!/bin/sh

i=1

while [ "$i" -le 5 ]
do
  echo $i
  i=$((i + 1))
done
```

Make the script executable and run it as follows:

```
./while_1
```

and note the output that you get.

Example 2

Create a file called `while_2` and type the following:

```
#!/bin/sh

#
# Print command line arguments one per line
#

while [ "$#" -ne 0 ]
do
  echo "$1"
  shift
done
```

Make the script executable and run it as follows:

`./while_2 a b c`

and note the output that you get.

0.6 The `until` Command

The `while` command continues execution as long as the command listed after the `while` returns a zero exit status. The `until` command is similar to the `while`, only it continues execution as long as the command that follows the `until` returns a nonzero exit status. As soon as a zero exit status is returned, the loop is terminated. Here is the general format of the `until`:

```
until commandt
do
  command
  command
  ...
done
```

Let's see how to use it by exploring some examples.

Example 1

Create a file called `until_1` and type the following:

```
#!/bin/sh

#
# Wait until a specified user logs on
#

if [ "$#" -ne 1 ]
then
    echo "Usage: mon user"
    exit 1
fi

user="$1"

#
# Check every minute for user logging on
#

until who | grep "^$user " > /dev/null
do
  sleep 60
done

#
# When we reach this point, the user has logged on
#

echo "$user has logged on"
```

The script `until_1` checks on a user provided as an argument if he/she is logged on to the system or not. If still not logged on, the script sleeps for `60` seconds before checking another time. That

behavior gets repeated until the user logs on.

The above script assumes you are behind your screen and are checking if a message gets displayed every now and then. In the below example, we're presenting an improved version of that script. The script is going to notify you optionally by email when a user gets logged on to the system.

Example 2

Create a file called `until_2` and type the following:

```
#!/bin/sh

#
# Wait until a specified user logs on
#

if [ "$1" = -m ]
then
    mailopt=TRUE
    shift
else
    mailopt=FALSE
fi

if [ "$#" -eq 0 -o "$#" -gt 1 ]
then
    echo "Usage: until_2 [-m] user"
    echo "-m means to be informed by mail"
    exit 1
fi

user="$1"

#
# Check every minute for user logging on
#

until who | grep "^$user " > /dev/null
do
  sleep 60
done

#
# When we reach this point, the user has logged on
#

if [ "$mailopt" = FALSE ]
then
    echo "$user has logged on"
else
    echo "$user has logged on" | mail hanna
fi
```

In the above example, we assume that `hanna` is the current user of the script. Note that the option `-m` has been added to allow the current user of the script (`hanna`) to be able to state if he wants to be notified by email or not.

0.7 Breaking Out of a Loop - the `break` command

Sometimes you may want to make an immediate `exit` from a loop. To just exit from the loop (and not from the program), you can use the `break` command, whose format is simply:

```
break
```

When the `break` is executed, control is sent immediately out of the loop, where execution then continues as normal with the command that follows the `done`.

The `break` command is often used to exit from these sorts of infinite loops, usually when some error condition or the end of processing is detected.

Example 1

Create a file called `break_1` and type the following:

```
#!/bin/sh

#
# The below code illustrates the use of the break command
#

while true
do
  cmd="$1"

  if [ "$cmd" = quit ]
  then
      break
  else
      echo "$cmd"
      sleep 1
  fi
done
```

Make the script executable and run it as follows:

```
./break_1 12
```

Do another run as follows:

```
./break_1 quit
```

Note the output that you get in each case.

Note that the above example has no useful application but to illustrate the usage of the `break` command.

Note

If the `break` command is used in the form:

```
break n
```

the `n` innermost loops are immediately exited.

Example 2

Create a file called `break_2` and type the following:

```
#!/bin/sh

#
```

```
# The below code is incomplete. It just illustrates on how to use
# the break command
#

for file
do
  ...
  while [ "$count" -lt 10 ]
  do
    ...

    if [ -n "$error" ]
    then
       break 2
    fi
    ...
  done
  ...
done
```

Please note that the above code is incomplete. It is provided to illustrate the usage of `break n` command.

In the above example, both the `while` and the `for` loops will be exited if variable `$error` is nonnull.

0.8 Skipping the Remaining Commands in a Loop - the `continue` Command

The `continue` command is similar to `break`, only it doesn't cause the loop to be exited, but the remaining commands in the loop to be skipped. Execution of the loop then continues as normal. Like the `break`, an optional number can follow the `continue`, so

```
continue n
```

causes the commands in the innermost `n` loops to be skipped; but execution of the loops then continues as normal.

Example

Create a file called `continue_1` and type the following:

```
#!/bin/sh

#
# The below illustrates the use of the continue command
#

for file
do
  if [ ! -e "$file" ]
  then
      echo "$file not found!"
      continue
  fi
#
```

```
# Process the file
#
echo $file

done
```

Make the script executable and run it twice, one by providing an existing file name as argument and the second time by providing a non-existant file name as argument.

Note the output that you get in each case.

0.9 Typing a Loop on One Line

If you would like to write loops on one line rather on multiple lines (to gain in visibility and have your script less lines of code), you can use the following shorthand notation to type the entire loop on a single line: Put a semicolon after the last item in the list and one after each command in the loop. Don't put a semicolon after the `do`.

Example

The below code

```
#!/bin/sh
for i in 1 2 3 4
do
  echo $i
done
```

can be written as:

```
#!/bin/sh
for i in 1 2 3 4; do echo $i; done
```

0.10 The `getopts` Command

It happens a lot that shell scripts are being passed arguments before they start running. The more arguments you provide, the more useful will shell scripts become (and more sophisticated as well).

The problem with arguments is that you need to pass them in a certain order so that a shell script knows how to deal with them. If the arguments are entered in different orders, the script might not behave the way you want. This is where the command `getopts` gets handy.

The general format of the command is:

```
getopts options variable
```

The following comments should be made regarding command `getopts`:

- The `getopts` command is designed to be executed inside a loop. Each time through the loop, `getopts` examines the next command line argument and determines whether it is a valid option. This determination is made by checking to see whether the argument begins with a minus sign and is followed by any single letter contained inside options. If it does, `getopts` stores the matching option letter inside the specified `variable` and returns a zero exit status.
- If the letter that follows the minus sign is not listed in options, `getopts` stores a question mark inside `variable` before returning with a zero exit status. It also writes an error message to standard error.

- All options that do not require arguments can be *stacked*. For example, if a shell script called `test` requires the following 3 options: `-a`, `-i`, `-r`, it can be run as follows:

  ```
  ./test -a -i -r
  ```

  or can be run as follows:

  ```
  ./test -air
  ```

- To indicate to `getopts` that an option takes a following argument, you write a colon (`:`) character after the option letter on the `getopts` command line. For example, if the shell script `test` takes 2 argument, `-m` and `-t` and option `-t` requires an argument, `getopts` should be used as follows:

  ```
  getopts mt: option
  ```

  If `getopts` doesn't find an argument after an option that requires one, it stores a question mark inside the specified variable and writes an error message to standard error. Otherwise, it stores the actual argument inside a special variable called `OPTARG`.

- A special variable called `OPTIND` is used by `getopts`. This variable is initially set to one and is updated each time `getopts` returns to reflect the number of the next command-line argument to be processed.

Let's look at some examples[2]:

Example 1

Create a file called `getopts_1` and type the following:

```
#!/bin/sh

while getopts  "abc:" flag
do
  echo "$flag" $OPTIND $OPTARG
done
```

Make the script executable and run it as follows:

```
./getopts_1 -abc "foo"
```

You should get the following output:

```
a 1
b 1
c 3 foo
```

Example 2

Create a file called `getopt_2` and type the following:

```
#!/bin/sh

while getopts  "abc:def:ghi" flag
do
  echo "$flag" $OPTIND $OPTARG
done

echo "Resetting"
```

[2]http://aplawrence.com/Unix/getopts.html

```
OPTIND=1

while getopts  "abc:def:ghi" flag
do
  echo "$flag" $OPTIND $OPTARG
done
```

Make the script executable and run it as follows:

```
./getopts_2 -a -bc foo -f "foo bar" -h -gde
```

You should get the following output:

```
a 2
b 2
c 4 foo
f 6 foo bar
h 7
g 7
d 7
e 8
Resetting
a 2
b 2
c 4 foo
f 6 foo bar
h 7
g 7
d 7
e 8
```

If command `getopts` encounters an unwanted argument, the variable `$flag` will be set to `?` and an error message will be displayed.

For example, run the shell script `getopts_2` as follows:

```
./getopts_2 -a -c foo -l
```

You will get the following output:

```
a 2
c 4 foo
./getopts_2: illegal option -- l
? 5
Resetting
a 2
c 4 foo
./getopts_2: illegal option -- l
? 5
```

Equally, if you do not provide an argument to an option that requires one, the variable `$flag` will be set to `?` and an error message will be displayed.

For example, run the shell script `getopts_2` as follows:

```
./getopts_2 -a -c
```

You will get the following output:

```
a 2
./getopts_2: option requires an argument -- c
```

```
? 3
Resetting
a 2
./getopts_2: option requires an argument -- c
? 3
```

0.11 The `read` Command

The `read` command is used to read standard input. That is helpful when a shell script needs to be interactive with users. The general format of the `read` command is:

```
read variables
```

When this command is executed, the shell reads a line from standard input and assigns the first word read to the first variable listed in `variables`, the second word read to the second variable, and so on. If there are more words on the line than there are variables listed, the excess words get assigned to the last variable.

If no user input is provided, a shell script halts until an input has been provided.

Example 1

Create a text file called `read_1` and type the following:

```
#!/bin/sh

#
# Copy a file
#

if [ "$#" -ne 2 ]
then
   echo "Usage: mycp from to"
   exit 1
fi

from="$1"
to="$2"

#
# See if the destination file already exists
#

if [ -e "$to" ]
then
   echo "$to already exists; overwrite (yes/no)?"
   read answer

   if [ "$answer" != yes ]
   then
      echo "Copy not performed"
      exit 0
   fi
fi

#
```

```
# Either destination doesn't exist or "yes" was typed
#

cp $from $to # proceed with the copy
```

The above example is a shell script that copies a source file into a destination file. However, it does some checkings before doing so, such as if the source and destination files exist or no, the number of arguments is correct and so on.

Note that the shell script will halt execution until an answer is provided by the user in case the destination file exists.

In the below example, we're presenting in advanced version of that script that shows how multiple files can be copied to a directory. The purpose is to show you that shell scripts can get quite complex if you intend to make them useful.

Example 2

Create a text file called `read_2` and type the following:

```
#!/bin/sh

#
# Copy a file -- final version
#

numargs=$#  # save this for later use

filelist=
copylist=

#
# Process the arguments, storing all but the last in filelist
#

while [ "$#" -gt 1 ]
do
  filelist="$filelist $1"
  shift
done

to="$1"

#
# If less than two args, or if more than two args and last arg
# is not a directory, then issue an error message
#

if [ "$numargs" -lt 2 -o "$numargs" -gt 2 -a ! -d "$to" ]
then
    echo "Usage: mycp file1 file2"
    echo " read_2 file(s) dir"
    exit 1
fi

#
# Sequence through each file in filelist
```

```
#

for from in $filelist
do
  #
  # See if destination file is a directory
  #

  if [ -d "$to" ]
  then
      tofile="$to/$(basename $from)"
  else
      tofile="$to"
  fi

  #
  # Add file to copylist if file doesn't already exist
  # or if user says it's okay to overwrite
  #

  if [ -e "$tofile" ]
  then
      echo "$tofile already exists; overwrite (yes/no)? \c"
      read answer

      if [ "$answer" = yes ]
      then
          copylist="$copylist $from"
      fi
  else
      copylist="$copylist $from"
  fi
done

#
# Now do the copy -- first make sure there's something to copy
#

if [ -n "$copylist" ]
then
    cp $copylist $to # proceed with the copy
fi
```

Make the script executable and run it as follows:

`./read_2`

Note the usage of the command `basename` in the above script. Its purpose is to strip directory and suffix from filenames. As an example, run the following command:

```
basename /home/user
```

and check on the output that you get. Try as well to run the command `dirname` whose purpose is to strip the last component from a file name:

```
dirname /home/user
```

and check on the output that you get. You will notice that the commands `basename` and `dirname` complement each other.

0.12 The `printf` command

Although `echo` is adequate for displaying simple messages, sometimes you'll want to print formatted output: for example, lining up columns of data. Unix systems provide the `printf` command. You should be familiar with it since you've seen it in the C programming language.

The general format of the `printf` command is:

```
printf "format" arg1 arg2 ...
```

where `format` is a string that describes how the remaining arguments are to be displayed.

Example

Type the following command on your shell:

```
printf "This is a number: %d\n" 10
```

The shell should print the following:

```
This is a number: 10
```

The command `printf` takes different conversion characters other than the `%d`. The below table summarizes these specification characters.

Character	Use for Printing
`d`	Integers
`u`	Unsigned integers
`o`	Octal integers
`x`	Hexadecimal integers, using a-f
`X`	Hexadecimal integers, using A-F
`c`	Single characters
`s`	Literal strings
`b`	Strings containing backslash escape characters
`%`	Percent signs

Examples and their output

```
printf "The octal value for %d is %o\n" 20 20
The octal value for 20 is 24

printf "The hexadecimal value for %d is %x\n" 30 30
The hexadecimal value for 30 is 1e

printf "The unsigned value for %d is %u\n" -1000 -1000
The unsigned value for -1000 is 4294966296
```

```
printf "This string contains a backslash escape: %s\n" "test\nstring"
This string contains a backslash escape: test\nstring

printf "This string contains an interpreted escape: %b\n" "test\nstring"
This string contains an interpreted escape: test string

printf "This string contains an interpreted escape: %b\n" "test\nstring"
This string contains an interpreted escape: test string

printf "A string: %s and a character: %c\n" hello A
A string: hello and a character: A

printf "Just the first character: %c\n" abc
a

printf "%+d\n%+d\n%+d\n" 10 -10 20
+10
-10
+20

printf "% d\n% d\n% d\n" 10 -10 20
 10
-10
 20

printf "%#o %#x\n" 100 200
0144 0xc8

printf "%20s%20s\n" string1 string2
           string1             string2

printf "%-20s%-20s\n" string1 string2
string1           string2

printf "%5d%5d%5d\n" 1 10 100
  1   10  100

printf "%5d%5d%5d\n" -1 -10 -100
 -1  -10 -100

printf "%-5d%-5d%-5d\n" 1 10 100
1  10   100

printf "%.5d %.4X\n" 10 27
00010 001B

printf "%.5s\n" abcdefg
abcde

printf "%.5s\n" abcdefg
abcde

printf "%12s%10.2s\n" "test one" "test two"
  test one        te
```