Shell Script Programming

Tutor: Ken Marr

Overview

This course gives an introduction to the writing and testing of both Linux and UNIX shell scripts using the Bash shell and to the advanced use of utilities such as sed and awk. Much of the course is suitable as preparation for the Linux Professional Institute (LPI) examinations.

Although based on CentOS or Ubuntu Linux the commands and principles learnt may be applied to all other versions of Linux and Unix.

Objectives

At the end of this course the delegate will be able to:

- write and test simple shell scripts using parameters
- use special characters and user and system variables
- build and test menu scripts
- use selection and loop commands such as *if*, *while* and *for*
- call a sub-script from within a script
- create and use functions

Schedule

The times shown may be changed as agreed with the tutor:

- Start time is 9.30am; end time is 4.15pm approx
- Break times, morning and afternoon
- Lunch will be for 1 hour at 12.30pm approx
- Please ensure that you **adhere** to the agreed times
- For an on-site course, **interruptions** should be kept to a minimum
- If you have a **mobile phone**, please turn it off now

Your tutor will point out the location of the following:

- The men's and women's toilets
- The fire exit and the action to be taken in the event of a fire

Introductions

Before the course begins, your tutor will ask each delegate to introduce themselves to the rest of the class.

In your introduction, please include the following:

- Name and company
- Job title and responsibilities
- Previous experience with the software to be used

Also consider answers to the following questions:

- I have come on the course because?
- What would I like to gain from the course?
- What would I like to be able to do by the end of the course?

Pre-requisites

This course is only suitable for those who have a working knowledge of Linux or Unix or have attended the Introduction to Linux or Unix course.

Responsibilities

The course is divided between lecture, demonstrations and exercises. The idea behind this is three fold. The delegate will:

- listen and learn
- see and understand
- do and remember

There are many exercises and workshops throughout this course. Please note the following:

- COOPERATION is encouraged, competition is not
- all the exercises may be undertaken in PAIRS
- delegates may, if they prefer, work TWO to a terminal
- if sharing try to achieve a 50% - 50% SPLIT at the keyboard

During the course, in order to aid the learning experience, the delegates should endeavour to:

- try out the course EXAMPLES
- attempt to complete each EXERCISE
- EXPERIMENT and try out new things
- not be afraid to ask QUESTIONS
- CONFER with their neighbours
- LEAN and be leaned on; the best way to learn is to help others
- learn by your (and their) MISTAKES
- ENJOY themselves!

Content

SHELL PROGRAMMING..7

SPECIAL CHARACTERS AND VARIABLES..16

CREATING MENUS..24

SELECTION..38

ITERATION, LOOPING...55

SUB-SCRIPTS AND FUNCTIONS...66

Shell Programming

Objectives

At the end of this section the delegate will be able to:

- create and test script files
- pass parameters to a script file
- prompt the user for input

Creating Shell Scripts

A shell script is simply an executable file containing a valid set of UNIX/Linux commands.

The steps involved in creating a shell script are as follows:

- test the individual commands interactively
- use an editor (*vi, vim, nano* or *gedit*) to enter the commands into a file
- consider using a file naming standard such as *scriptname.sh*
- include comments, lines beginning with a #
- include blank lines to help readability
- make the script executable
- test the script

Testing the Command

For example, the following command displays a message (use banner on Unix):

```
figlet  hello world
```

If this command is not available, use the echo command:

Creating the Script File

Use an editor to enter the following into a file called *hello*:

```
# A first shell script to display a message

figlet hello world
```

Testing the Script

The script can be tested as follows:

```
bash  hello
```

or

```
.  hello
```

An attempt can be made to execute the script by typing its full path name but this returns an error:

```
/home/train1/hello
```

or

```
./hello
```

Adding Execute Permissions

This is because execute permissions must first be added to the script before it can be tested:

```
chmod  755  hello

./hello
```

When the script runs, it forks a new shell and then control is returned to the login shell.

The PATH Variable

An attempt can now be made to execute the script by typing just its name as you would a command:

```
hello
```

However, this will only work if the current working directory is included in the PATH variable. This variable can be viewed as follows:

```
echo  $PATH
```

If the current working directory is not present, the PATH variable may be amended to add the directory:

```
export  PATH=$PATH:.
```

And then the script may be run:

```
hello
```

Login Files

Note that if the PATH variable is amended as above this will only take effect for the current session.

A better way is to amend the PATH variable in the *.bashrc* file so that it takes effect the next time the user logs in.

. - dot Command

As an alternative to logging out and in again after amending one of the login files, the file may be tested thus:

```
            . .bashrc
```

The dot (.) command runs the contents of the file in the current shell.

Exercise

1. Create and test a new shell script called *fcount,* which will count the number of files in the current working directory.

   For the moment, do not attempt to display the name of the directory. This will be added in a later exercise.

   Hint: use the commands *ls*, *wc -l* and a pipe.

Passing Parameters

Parameters can be passed to a script file on the command line. They are accessed within the script as $1, $2, …….$9, ${10} etc.

For example, the following file accepts 4 parameters:

```
# A first shell script to display a message

echo  hello  $1  $2  $3  $4
```

This script can be passed up to 4 names as parameters after the script name on the command line:

```
hello  dave  and  jane
```

Other parameters available include $0, $* and $#. In this case:

- $0 is the name of the shell script
- $# is the number of parameters
- $* is a list of all parameters

Note that parameters may also be passed to an alias.

Exercise

1. Copy the script file *param* from the directory */scr*. This contains lines similar to the following:

```
# A shell script which accepts parameters
echo  You entered the words $1 $2 $3 $4 $5

echo  This script is called $0

echo  $0 was supplied with $# parameters

echo  The parameters are: $*
```

2. Test the script by passing names as parameters after the script file on the command line. For example:

```
            param  a  b  c  d  e  f
 or
            param  ham  egg  sausage
```

   Try running the script with 5 parameters, less than 5 and more than 5 parameters.

3. Amend the script *fcount* to accept a command line parameter as the name of the directory to be counted. Test the script using the directories /*tmp*, /*bin* and *temp*:

```
            fcount  /tmp
```

   For the moment, do not attempt to display the name of the directory. This will be added in a later exercise.

read Command

Input can also be stored in user created variables and these can be populated using the read command.

For example, at the command prompt try the following and respond with your name:

```
read  name
```

Now enter:

```
echo  $name
```

Note that the $ sign is NOT used in the read command but is used when displaying the variable.

read in Scripts

Input can also be passed to a script by means of the read command. For example:

```
# A shell script which prompts for user input

echo  Please enter your name:
read  ans
echo  Your name is $ans
```

When the script is run, it will prompt the user to enter their name.

Exercise

1. Enter the above into a file called *name* and test the new script.

2. Amend the script *fcount* to prompt for the name of the directory to be counted.

Tidying The Output

The following special characters may be used in most shells to format the output from the *echo* command:

\c	keep cursor at the end of the line
\n	newline
\t	tab

The script may be amended thus:

```
# A shell script which prompts for user input

clear
echo  -e  “\n\tPlease enter your name: \c”
read  ans
echo  -e  “\tYour name is $ans”
```

Note that on Linux the option -e must be passed to the echo command in order for these special characters to be used:

Choosing the Shell

There are significant differences between the Bash shell and other shells (such as the C shell) in that many Bash shell scripts will not run in other shells and vice-versa. However, a script can be made to run in a particular shell. To find the path of the shell use:

```
which  bash
```

The following script will always fork and run in a Bash shell even if it was executed from another shell such as the C shell:

```
#!/bin/bash
clear
echo  -e   “\n\tPlease enter your name: \c”
read  ans
echo  -e   “\tYour name is $ans”
```

Exercise

1. Amend the script *fcount* and tidy up the output from the script using the above syntax.

Debugging the Script

If the script does not work properly, it can be run with the trace options of the *set* command and passed as a parameter to a shell as follows:

```
bash  -xv  hello
```

Review Exercise

1. The command *stty* may be used to suppress the input from a user when they are prompted by the *read* command.

   For example, to suppress output to the screen, use the command:

   ```
   stty  -echo
   ```

   To turn output to the screen back on, use the command:

   ```
   stty  echo
   ```

2. Create a new shell script called *pcheck* which will simply prompt the user to enter both a username and a password. However, the password entered should NOT be displayed on the screen.

   Hint: use the command *stty* either side of the *read* command.

   If you have time:

3. Create and test a script called *backup.sh* which will backup all of your files to disc using the *tar* command.

   Hint: use a command of the form:

   ```
   tar  czvf  /backup/train$(date  '+%m%d%H%M').tar.gz  .
   ```

4. Use the *crontab* command to schedule a job to run the backup script during the next break.

   Hint: use the command *crontab -e* to schedule the backup

5. Once the backup script has worked correctly, remove it with the command

   ```
   crontab  -r
   ```

Special Characters and Variables

Objectives

At the end of this section the delegate will be able to:

- describe the use of the different quotation marks
- describe the use of the different sets of brackets
- describe and display various system variables
- create and display user variables
- create both local and global variables

Quotation Marks

The three quotation marks available may be used around string variables but the different quotation marks behave in different ways.

Quotes - Double, Single

Try the following:

```
echo  *

echo  “*”

echo  ‘*’
```

Both single and double quotes protect the shell from special characters such as * and ?.

An example of this is the command:

```
find  .  -name  p*  -print
```

which may only work if quotes are used:

```
find  .  -name  “p*”  -print
```

Quotes - Single

Now try:

```
echo  $PATH

echo  “$PATH”

echo  ‘$PATH’
```

Only single quotes protect the shell from the special character $.

An example of this is the command:

```
grep  ‘x$’  linux
```

When using quotes in a script, use double quotes around a variable:

```
echo -e  “\n\tYour name is $ans”
```

and NOT single quotes:

```
echo  'Your name is $ans'
```

Escape Character

The escape character may also be used to protect the shell from the special characters * and $. Try the following:

```
echo  $HOME

echo  \$HOME

echo  *

echo  \*
```

Examples of this used with a command include:

```
\ls

find  .  -name  “p*”  -ok  rm  {}  \;
```

Quotes - Back

Back quotes behave in a different way to the other quotation marks. Try the following:

```
echo  date

echo  “date”

echo  ‘date’

echo  `date`
```

Back quotes expect the string to be a valid command or shell script file.

The following is an example of the use of back quotes with the *echo* command:

```
echo  “Number of users = ”  `who  |  wc  -l`
```

Using $(…)

As an alternative to back quotes the following syntax may be used:

```
echo  “Number of users = ”  $(who  |  wc  -l)
```

Exercise

1 Amend the script file *fcount* to display the number of files in a given directory in a message of the form:

```
Number of files in DIR  =  N
```

where DIR is the name of the directory and N is the number of files.

Hint: use the *echo*, *ls* and *wc* commands and $(…).

Variables - System

The shell uses important environment variables such as PATH, TERM and HOME. These are usually set at log in time by commands in the **.**profile file or /etc/profile file.

These variables can be displayed using the *set* or *env* commands:

```
set | more
```

They may also be displayed using the echo command, but a variable displayed in this way must be prefixed by a $ sign:

```
echo  PATH

echo  $PATH

echo  $HOME
```

Variables - Special

There are many other special built in system variables. For example, the variable $? holds the return code of the last command. This will be zero if the command completed OK but non-zero if the command failed:

```
rm  text1

echo  $?

rm  text1

echo  $?
```

The system variable $$ holds the PID number of the current shell:

```
ps

echo  $$
```

Other built in variables are covered later in the course.

Variables - User

Users can create and use their own shell variables.

For example, to create a variable in the current shell called *name* assigned the value steve:

```
name="steve"
```

Note that there are no space characters around the = sign. A variable may also be set to the output from a command:

```
date=$(date)
```

These variables may be displayed using the echo command, but again a variable displayed in this way must be prefixed by a $ sign:

For example, to display the text name:

```
echo  name
```

To display the value of the variables:

```
echo  $name
echo  $date
```

Variables - unset

The command *unset* may used to set the value of a variable to null:

```
unset  name
echo  $name
```

Variables in Scripts

The parameters $1, $2 etc are read only variables. User Variables may be stored and used within a script file and may be set to a character string or to a passed parameter, such as $1, $2, $3 etc.

```
# A shell script which displays a variable
user="$1"

echo  “User =  " $user
```

Variables - Local

A variable is LOCAL to the shell in which it was created.

For example, the following creates and displays a LOCAL variable:

```
user="steve"
echo  $user
```

Variables - Global

However, a variable may be made GLOBAL and available to other programs, or sub shells, by being exported via the *export* command.

This example creates and displays a GLOBAL variable which will be available in all sub shells:

```
user="steve"
export  user
echo  $user
```

or

```
export user="steve"
echo  $user
```

Variable Constructs - { }

The curly braces also have another use and can be used with five special variable constructs.

${variable}

Here they are used to separate a variable from any adjoining text.

For example:

```
name=”ken”

echo  $names

echo  ${name}s
```

${variable:-value}

If the variable is null, use the value instead.

For example:

```
echo  ${name:-alan}

unset  name

echo  ${name:-alan}
```

The following will set the variable *user* to the value *steve* if the user does not enter a value:

```
# A shell script which displays a variable
echo  -e  “\n\tPlease enter a user name:\c”
read  user
user="${user:-steve}"
```

Exercise

1. Amend the script file *fcount* to display the current working directory if no directory name is entered.

   Hint: use the above syntax to set a default value of $PWD

Creating Menus

Objectives

At the end of this section the delegate will be able to:

- use the command case
- create and test a simple menu script
- use the command select to create a menu

case - Condition Test

The system supports the *if* command which is covered later in the course. As an alternative, the command *case* may be used to provide a multiple test statement.

For example, the following script runs either the *who*, *ps, ls* or *fcount* command depending on the number entered:

```
# A shell script which tests user input
echo  -e   "Please enter selection:\c "
read ans

case  "$ans"  in
        1) who          ;;
        2) ps           ;;
        3) ls           ;;
        4) ./fcount     ;;
        *)echo 'Enter valid selection'  ;;
esac
```

Note that each test must end with a closing bracket and that each command must end with two semi-colons, the characters **;;**.

The final test, which uses an *, traps any other input.

Menu Creation

The command *case* may be used to create a simple menu. For example:

```
                # Simple menu script

                echo -e   "       Menu

                        1: List users
                        2: List processes
                        3: List files
                        4: Count files

                        Enter Selection: \c "
                read ans

                case  "$ans"  in
                        1) who                  ;;
                        2) ps                   ;;
                        3) ls                   ;;
                        4) ./fcount             ;;
                        *)echo 'Enter valid selection'  ;;
                esac
```

Exercise

1̃ Copy the script *menu* from the directory */scr*. This contains lines similar to those above.

2̃ Test the menu script by entering different options.

while, until - Iteration

The above script may be amended to include an infinite loop as follows. Here the command *clear* is used to clear the screen for each iteration:

```
# Simple menu script
while true
do
        clear
        echo -e   "         Menu

                1: List users
                2: List processes
                3: List files
                4: Count files

        Enter Selection:\c "
        read ans
        case  "$ans"  in
                1) who                  ;;
                2) ps           ;;
                3) ls           ;;
                4) ./fcount     ;;
                *)echo 'Enter valid selection'  ;;
        esac
done
```

The loop may be ended by use of CONTROL & C.

The *until* loop is similar to while loop but loops until the end condition becomes true.

Exercise

ì Amend the script file *menu* to use a loop and the clear command and test the new menu.

The above script runs OK but there is a problem with the output!

sleep - Holding the Output

The script runs OK but the output is not held on the screen.

The command *sleep* may be used to cure this problem:

```
# Simple menu script
while true
do
        clear
        echo -e   "         Menu

                1: List users
                2: List processes
                3: List files
                4: Count files

         Enter Selection:\c "
         read ans
         case  "$ans"  in
                1) who          ;;
                2) ps           ;;
                3) ls           ;;
                4) ./fcount     ;;
                 *)echo 'Enter valid selection'  ;;
        esac
        sleep 2
done
```

Exercise

ῗ Amend the script file *menu* to use the sleep command and test the new menu.

read - Holding the Output

As an alternative to the sleep command, a dummy *read* command may also be used to cure this problem:

```
# Simple menu script
while true
do
        clear
        echo -e   "         Menu

                1: List users
                2: List processes
                3: List files
                4: Count files

        Enter Selection:\c "
        read ans
        case  "$ans"  in
                1) who          ;;
                2) ps           ;;
                3) ls           ;;
                4) ./fcount     ;;
                *)echo 'Enter valid selection'  ;;
        esac
        echo “Please press return to continue”
        read ans
done
```

Exercise

ì If you prefer, amend the script file *menu* to use a dummy read command and test the new menu.

exit - Ending the Script

As the script stands, the only way to end the loop is by use of CONTROL & C. However, the command *exit* may be used to end the script as follows:

```
# Simple menu script
while true
do
        clear
        echo -e   "         Menu

                1: List users
                2: List processes
                3: List files
                4: Count files
                Q: Exit

                Enter Selection:\c "
        read ans
        case  "$ans"  in
                1) who          ;;
                2) ps           ;;
                3) ls           ;;
                4) ./fcount     ;;
                [Qq]) exit      ;;
                *)echo 'Enter valid selection'  ;;
        esac
        echo "Please press return to continue"
        read ans
done
```

Exercise

1 Amend the script file *menu* to use the exit command to end the script and test the new menu.

break - Ending the Loop

As an alternative to the *exit* command, the command *break* may be used to end the loop and then the script as follows:

```
# Simple menu script
while true
do
        clear
        echo -e   "         Menu

                1: List users
                2: List processes
                3: List files
                4: Count files
                B: Break
                Q: Exit

                Enter Selection:\c "
        read ans
        case  "$ans"  in
                1) who          ;;
                2) ps           ;;
                3) ls           ;;
                4) ./fcount     ;;
                [Bb]) break  ;;
                [Qq]) exit      ;;
                *)echo 'Enter valid selection'  ;;
        esac
        echo “Please press return to continue”
        read ans
done
echo “End of menu script”
```

Note that if the option Q is chosen, the script ends but if the option B is chosen the loop ends and the message *End of menu script* is displayed before the script ends.

Exercise

1 Amend the script file *menu* to use the break command to end the loop and test the new menu.

Hidden Option - Forking a shell

A hidden option, to fork a Korn or Bash shell, may be added to the menu as follows:

```
# Simple menu script
while true
do
        clear
        echo -e   "         Menu

                1: List users
                2: List processes
                3: List files
                4: Count files
                B: Break
                Q: Exit

                Enter Selection:\c "
        read ans
        case  "$ans"  in
                1) who          ;;
                2) ps           ;;
                3) ls           ;;
                4) ./fcount     ;;
                sh) ksh         ;;
                [Bb]) break     ;;
                [Qq]) exit      ;;
                *)echo 'Enter valid selection'  ;;
        esac
        echo "Please press return to continue"
        read ans
done
echo "End of menu script"
```

Note that if the hidden option, which is not displayed, is chosen, the menu is suspended and a new shell is invoked. This is a better idea than using a single character such as K or B, since this may be inadvertently typed in by the user.

The *exit* command may be used to return to the menu.

Exercise

1. Amend the script file *menu* to include a hidden option which forks a new shell.

trap - Interrupts

The command *trap* may be used to trap interrupts such as the use of CONTROL & C. This is done by trapping a signal. Signals which can be trapped include:

•	2	-	user break out, CONTROL & C
•	15	-	normal kill
•	20	-	move job to background, CONTROL & Z

The script may be amended to include a trap as follows. This will trap a user break out, the kill command and CTRL & Z:

```
# Simple menu script
trap  'echo -e "\nPlease enter Q to quit" '  2  15  20
while true
do
        clear
        echo -e   "          Menu

                1: List users
                2: List processes
                3: List files
                4: Count files
                B: Break
                Q: Exit

                Enter Selection:\c "
        read ans
        case  "$ans"  in
                1) who          ;;
                2) ps           ;;
                3) ls           ;;
                4) ./fcount     ;;
                sh) ksh         ;;
                [Bb]) break     ;;
                [Qq]) exit      ;;
                *)echo 'Enter valid selection'  ;;
        esac
        echo "Please press return to continue"
        read ans
done
echo "End of menu script"
```

Exercise

1 Amend the script file *menu* to include the above trap and test the menu by trying to break out with CONTROL & C and by using the kill command.

trap - Continue

The script runs OK but the messages *Enter valid selection* and *Please press return to continue* still appear.

These may be suppressed by including a null input, signified by **two single quotes**, in the case statement and the command *continue*, which forces control back to the top of the loop:

```
# Simple menu script
trap  ‘echo -e " \nPlease enter Q to quit" ‘  2  15  20
while true
do
        clear
        echo -e   "         Menu

                1: List users
                2: List processes
                3: List files
                4: Count files
                B: Break
                Q: Exit

                Enter Selection:\c "
        read ans
        case  "$ans"  in
                1) who          ;;
                2) ps           ;;
                3) ls           ;;
                4) ./fcount     ;;
                ‘’) continue    ;;
                sh) ksh         ;;
                [Bb]) break     ;;
                [Qq]) exit      ;;
                *)echo 'Enter valid selection'  ;;
        esac
        echo “Please press return to continue”
        read ans
done
echo “End of menu script”
```

Note that if a null command is needed, the character **:** may be used.

Exercise

- Amend the script file *menu* to include the above amendment and test the menu again by trying to break out with CONTROL & C.

tput - Highlighting Text

The command *tput* may be used to highlight text on the screen. Depending on the version in use it can produce effects such as bold and inverse video. To demonstrate this the script may be amended as follows:

```
# Simple menu script
INV=$(tput  smso)
NOR=$(tput  rmso)
BOL=$(tput  bold)
trap ‘echo -e “\n${BOL}Please enter Q to quit${NOR}" ’  2  15  20
while true
do
        clear
        echo -e   "        ${INV}Menu${NOR}

                1: List users
                2: List processes
                3: List files
                4: Count files
                B: Break
                Q: Exit

                Enter Selection:\c "
        read ans
        case  "$ans"  in
                1) who          ;;
                2) ps           ;;
                3) ls           ;;
                4) ./fcount     ;;
                ‘’) continue    ;;
                sh) ksh         ;;
                [Bb]) break     ;;
                [Qq]) exit      ;;
                *)echo 'Enter valid selection'  ;;
        esac
        echo “Please press return to continue”
        read ans
done
echo “End of menu script”
```

Note the use of braces, { }, used to separate the name of the variable from the text. To reset the screen at the command prompt use the *reset* command.

Exercise

1̃ Amend the script file *menu* to highlight some text and test the new menu.

Positioning the Cursor - backtab \b

As well as \c, the special characters \b, backtab, may be used to position the cursor:

```
# Simple menu script
INV=$(tput  smso)
NOR=$(tput  rmso)
BOL=$(tput  bold)
trap ‘echo -e “\n${BOL}Please enter Q to quit${NOR}" ’  2  15  20
while true
do
        clear
        echo -e   "         ${INV}Menu${NOR}

                1: List users
                2: List processes
                3: List files
                4: Count files
                B: Break
                Q: Exit

                Enter Selection [  ]\b\b\c"
        read ans
        case  "$ans"  in
                1) who          ;;
                2) ps           ;;
                3) ls           ;;
                4) ./fcount     ;;
                 “) continue    ;;
                sh) ksh         ;;
                [Bb]) break     ;;
                [Qq]) exit      ;;
                *)echo 'Enter valid selection'  ;;
        esac
        echo “Please press return to continue”
        read ans
done
echo “End of menu script”
```

Exercise

ĩ Amend the script file *menu* as above to position the cursor and test the new menu.

exec - Running the Script

The exec command runs a script but replaces the current shell with a new shell:

```
exec  menu
```

This is illustrated by the following exercise.

Exercise

1̃ To invoke the menu on logon, include the following call:

```
./menu
```

as the last line in the file *.bashrc*.

2̃ As an alternative, invoke the menu on logon with the following call:

```
exec  ./menu
```

Now quit from the menu and note the difference!

3̃ Use the hidden option of the menu to fork a new shell and edit the *.bashrc* file to remove the call to the menu.

Note that if your hidden option is the **bash** shell, this will again fork a new menu!!

Selection

Objectives

At the end of this section the delegate will be able to:

- use the condition command if to test file attributes
- use the command if to test strings and numbers
- use the command if to test other commands
- use the more advanced options of the case command

if - Conditional Testing

The following script *chx* adds execute permissions to up to 3 files whose names are given as parameters:

```
# A shell script which adds execute permissions
chmod  +x   $1  $2  $3
```

However, if a 4th file name is passed the script will not work correctly.

A solution is to amend the script as follows so that it adds execute permissions to however many file names are given as parameters:

```
# A shell script which adds execute permissions
chmod  +x   $*
```

Now, consider what happens if the script is run with no parameters as follows; an error occurs:

```
chx
```

To correct this error, the script may be amended to include the commands *if* and *test* to allow input to be tested to determine its contents.

Here, the comparison operator -z is used to test for a **null** or empty string.

For example:

```
# A shell script which adds execute permissions
if  test  -z  "$1"
then
        echo "No filenames supplied"
else
        chmod  +x  $*
fi
```

Exercise

1. Create and run the script file *chx* as above to test for a null parameter.

This command returns True or False and uses the variable $? which holds the return code of the test. This will be zero if the test completed OK but non-zero if the test failed.

if - File Attributes

The same commands allow the attributes of a file to be tested.

The following are some of the file conditions that may be checked:

-e	object exists, may be a file **or** directory
-f	object exists and is a regular **file**
-d	object exists and is a **directory**
-s	object has a **size** greater than zero
-r	has **read** permissions
-w	has **write** permissions
-x	has **execute** permissions

For example:

```
# A shell script which tests files

echo  -e  "\n\tPlease enter a file name:\c"
read ans

if  test   -f  "$ans"
then
        echo  "$ans is a file"
else
        echo "Error"
fi
```

if - Login Files

The command *if* may be used in the file *.profile* to ensure that the same commands are executed in the *.bashrc* file on both login and when a shell is forked:

```
# Execute .bashrc if it exists
#
if  test  -f  ./.bashrc
then
        . ./.bashrc
fi
```

if - Multiple Tests, elif

Multiple conditions in one if statement may be coded as follows:

```
if  test   -d  "$ans"
then
        echo  "$ans is a directory"
elif  test   -f  "$ans"
then
        echo  "$ans is a file"
else
        echo "Error"
fi
```

if - NOT Condition

With both file and permissions test, the NOT condition may be tested by using the ! character.

In the following example, note the **space** between the ! and the -w:

```
# A shell script which tests files permissions

echo  -e  "\n\tPlease enter a file name:\c"
read ans

if  test   !  -w  "$ans"
then
        echo  "$ans has no write permissions"
else
        echo "Error"
fi
```

if - Multiple Tests, AND and OR

Multiple conditions may be tested in one test by using the following options:

-a	for an AND condition
-o	for an OR condition

For example:

```
# A shell script which tests files permissions

echo  -e  "\n\tPlease enter a file name:\c"
read ans

if  test  -d "$ans"  -a  -x  “$ans”
then
        echo  "$ans is a directory with execute permissions"

elif  test  -r  “$ans”  -o  -w  “$ans”
then
        echo  "$ans  has read or write permissions"
else
        echo "Error"
fi
```

Exercise

1. Create a new script *ftest*, which will prompt for a simple file name and test it as follows.

   If the file exists and is NOT empty, display the message "File is Not Empty".

      Hint: use the option *-f* to test for a file and *-s* to test if it has content.

   If the file exists and is empty, display the message "File is Empty".

   If the file does NOT exist, display the message "File Not Found".

2. Test the script by passing it an existing file, a file that does not exist and an empty file created using the *touch* command.

3. Remember, if the script does not work, debug it by running it with a new shell and the options -vx, for example:

```
bash  -vx  ftest
```

if - Multiple Tests, Nested

Multiple nested *if* tests may be coded as follows:

```
# A shell script which tests files permissions

echo  -e  "\n\tPlease enter a name:\c"
read ans

if  test   -d  "$ans"
then
        echo  "$ans is a directory"
else
        if  test   -f  "$ans"
        then
                echo  "$ans is a file"
        else
                echo "Error"
        fi
fi
```

if - Testing Strings

The commands *if* and *test* also allow input to be passed to a parameter and tested to determine its contents as a string.

The following may be used to compare strings:

```
=    !=    <    >    <=    >=
```

For example:

```
# A shell script which tests user input
echo  -e  "\n\tPlease enter a name:\c"
read ans

if  test  $ans   =   "Steve"
then
        echo "Correct"
else
        echo "Wrong"
fi
```

Note the **spaces** around the = sign.

This example uses the syntax *not equal*:

```
# A shell script which tests user input
echo  -e  "\n\tPlease enter a name:\c"
read ans

if  test  $ans   !=   "Steve"
then
        echo "Wrong"
else
        echo "Correct"
fi
```

if - Using Square Brackets

The following syntax replacing the command test with square brackets may be easier to remember.

```
# A shell script which tests user input

echo  -e  "\n\tPlease enter a name:\c"
read ans

if   [   $ans   =   "Steve"   ]
then
        echo "Correct"

elif   [   $ans   =   "steve"   ]
then
        echo "Correct"
else
        echo "Wrong"
fi
```

Note that there MUST be a space either side of each bracket used.

Multiple conditions may be also tested in one test using one set of square brackets, for example:

```
if  [  $ans  =  “steve”  -o  $ans  =  “Steve”   ]
```

Exercise

1̃ Create the new script file *testname*, based on the *name* script, and amend it as in the above example. Use your own name if you prefer.

2̃ Test the script file *testname* for null input by just pressing return.

If it does not work, debug it by running it with a new shell, for example:

```
bash  -xv  testname
```

if - Avoiding Null Input

The script works well if a name is entered but If the script is run and the return key is pressed, an error occurs. This is because the variable ans contains a NULL value: it is empty.

To avoid this, it is good practice to always place such a variable in double quotes.

```
# A shell script which tests user input

echo  -e  "\n\tPlease enter a name:\c"
read ans

if  [   "$ans"  =  "Steve"  ]
then
        echo "Correct"

elif  [   "$ans"  =  "steve"  ]
then
        echo "Correct"
else
        echo "Wrong"
fi
```

Exercise

1. Amend the script file *testname* as above and test it for null input.

if - Testing Numbers

The commands *if* and *test* also allow input to be tested to see if it contains a number.

The following may be used to compare numbers:

```
-eq  -ne  -lt  -gt  -le  -ge
```

For example, the following will return false:

```
if    test    "1"    =    "01"
```

but this syntax will return true:

```
if    test    "1"    -eq    "01"
```

This example script tests for a number entered:

```
# A shell script which tests numbers

echo  -e  "\n\tPlease enter a number:\c"
read ans

if   test   "$ans"   -lt   "100"
then
        echo  "$ans is less than 100"

elif   test   "$ans"   -gt   "100"
then
        echo  "$ans is greater than 100"
else
        echo  "$ans is equal to 100"
fi
```

Exercise

1. Amend the script *chx* and rather than testing $1for a null value, test the number of parameters passed and display a message if the number is zero.

   Hint: Test the variable $#, the number of parameters supplied.

Placing Spaces

It is very important that space characters are used correctly within a script file. For example, **do not** put spaces around the = sign when declaring a VARIABLE. Variables must be declared as follows:

```
user="steve"
```

Do put spaces around almost everything when using an IF statement.

For example:

```
if   test   $ans   =   "Steve"

if   test   -d   "$ans"   -a   !   -x   “$ans”

if   [   -d   "$ans"   -a   !   -x   “$ans”   ]
```

Read Command

Remember that the $ sign is NOT used in the read command but is used when displaying the variable. For example:

```
read ans

echo  "$ans"
```

case - Testing Names

The command *case* provides a multiple if statement and may also be used in order to test the spelling of a name entered:

```
# A shell script which tests user input
echo  -e  "\n\tPlease enter a name:\c"
read ans

case "$ans" in
        Steve)          echo "Correct"
        ;;
        steve)          echo "Correct"
        ;;
         *)             echo "Wrong"
        ;;
esac
```

Note that each test must end with a closing bracket and that each command must end with the characters **;;**.

The final test, which uses an *, traps any other input.

Different spellings of a name may be tested on one line by using the pipe symbol:

```
# A shell script which tests user input

echo  -e  "\n\tPlease enter a name:\c"
read ans

case "$ans" in
        Steve I steve I STEVE)     echo "Hi Steve”
        ;;
        Ann I ann I ANN)           echo "Hi Ann"
        ;;
         *)                        echo "Wrong"
        ;;
esac
```

The script may be further amended to test any spelling by using square brackets:

```
# A shell script which tests user input

echo  -e  "\n\tPlease enter a name:\c"
read ans

case "$ans" in
        [Ss][Tt][Ee][Vv][Ee])       echo “Hi Steve”
        ;;
        [Aa][Nn][Nn])               echo “Hi Ann”
        ;;
         *)                         echo "Wrong"
        ;;
esac
```

The following script validates the input by testing for a **single** character, input in either upper or lower case:

```
# A shell script which tests user input

echo  -e  "\n\tPlease enter your input:\c"
read ans

case "$ans" in
        [A-Z]  |  [a-z])       echo “Correct”
        ;;
        *)                     echo "Only 1 character allowed"
        ;;
esac
```

Exercise

1̃ Create a new script file called *casename,* based on the script *testname*, which will use a case statement as above to test your own name as input.

if - Testing Commands

The command *if* on its own, **without** the word test or the syntax using square brackets, allows the direct testing of a command.

The command returns True or False and uses the variable $? which holds the return code of the test. This will be zero if the command completed OK but non-zero if the command failed.

In this example any output from the command is redirected to /dev/null:

```
# A shell script which tests a command

echo  -e  "\n\tPlease enter a process name:\c"
read ans

if    ps l grep   "$ans" | grep  -v  grep  > /dev/null  2>&1
then
        echo  "\tThe process is still running"
else
        echo  "\tThe process has completed"
fi
```

Exercise

1. Copy the script */scr/pscheck* into your home directory and run it to see if a background process such as *sleep* is running.

2. Now run it and pass your login shell as the parameter.

Using Variables

Variables can be tested within a script file and can also be set to an initial value.

The following script is an example of how to set, test and display variables;

```
# A shell script which tests a variable

USER=”steve”
PASS=”smith”

echo "Please enter your name:"

read user

if  [  "$user" = "$USER"  ]
then
      echo "User valid"
else
      echo "User in error"
fi
```

Logical Constructs

The construct [[may be used in place of [and the two constructs && and || may be used to test the completion or otherwise of the proceeding command and conditionally execute a second command.

The exit status of the first command in $? is explicitly tested to achieve this.

&&	if command 1 works, execute command 2
||	if command 1 fails, execute command 2
[[ ]]	alternative to the test and [constructs

For example:

```
touch  test1

mv  test1  test2  2>/dev/null  &&  echo  file test1 moved

mv  test1  test2  2>/dev/null  &&  echo  file test1 moved

mv  test1  test2  2>/dev/null   ||   echo  file test1 not moved
```

These constructs can also be used in a script.

For example, the && construct may be used with the [[construct to execute a command if the previous command completed OK. The || construct may be used to display a message if the previous command fails:

```
# A shell script tests a file

echo -e "\n\tPlease enter a file name: \c"
read ans

if  [[   -f  $ans  ]]  &&  [[   -s  $ans  ]]
then
        echo  “$ans is not empty”
elif  [[  -f  $ans  ]]
then
        echo  "$ans is empty"
else
        echo  “$ans not found”
fi
```

Another way to avoid the problem of NULL input is to use the double square brackets syntax as above since here any syntax errors are ignored:

Note that this version of the IF statement does **not** support the use of the -a for AND or the -o for OR syntax.

Review Exercise

1. *Using variables and an if statement.*

   Amend the script file called *pcheck* as follows.

   Store both a valid user name and a valid password in variables within the script.

   Prompt the user to enter both a user name and a password. The password entered should NOT be displayed on the screen.

   Include an *if* statement to check the user name and password entered against the values stored in the two variables.

   Display either the message “User valid” or “User in error”.

2. *Testing a command with an if statement.*

   Create and test a script called *smess* which will take as input a user id.

   If the user is logged on then the file */scr/message* is displayed on their screen.

   If the user is not logged on then the file */scr/message* is mailed to the user.

   Hint: use the commands *who* and *grep* to check to see whether the user is logged on or not. Then redirect the file */scr/message* into the *write* or *mail* command.

   If you have time:

3. Amend the script file *smess* and rather than using the file */scr/message*, pass a message into the script as a parameter and either write or mail the message to the user.

   Hint: use the command *echo* and a pipe to send the message.

Iteration, Looping

Objectives

At the end of this section the delegate will be able to:

- use the let command to perform calculations
- use the while command in different forms to code a loop
- use the for command to code a loop
- use the shift command

let - Computations

The command *let* may be used to perform mathematical operations on integers only.

The operators used are: +, -, /, * and % for modulus.

For example:

```
let  w=2+5
let  x=10*3
let  ans=10%3

echo $w  $x  $ans
```

The variable created may be used with or without the $ sign:

```
echo w is $w
let  y=w+6;echo  $y
let  z=$w*4;echo  $z
```

This command may be used to increment a counter variable as follows:

```
count=0
let  count=count+1
echo $count
```

On some Linux servers this syntax also adds one to the counter variable:

```
count=0
let  count++
echo $count
```

() - Computations

The following syntax may be used in place of the *let* command:

```
((a=3+8))
((b=15*4))
((rem=38%5))

echo $a  $b  $rem

echo a is $a
((c=a+8));echo  $c
((d=$a*5));echo  $d
```

This may be used to increment a counter variable as follows:

```
count=0
((count=count+1))
echo $count
```

expr - Computations

The command *expr* may be used, for example if the *let* command is not available, to perform mathematical or comparison operations on integers.

It makes up in some part for most other shells inability to evaluate mathematical expressions directly.

The operators used are: +, -, /, *, and %, for example:

```
expr  2 + 5
expr 10 \* 3
expr 10 /  3
expr 10 % 3
```

Note the use of spaces and the escape character \.

Expr is normally used in conjunction with $(..) or back quotes. For example, to increment a counter use:

```
count=0
count=$(expr  $count + 1)
echo $count
```

while, until - Iteration

The commands *while* and *until* may also be used to perform a loop within a shell script.

The following script, */scr/stimes*, illustrates the use of the *let* command and calculates a 3 times table by default unless given a parameter:

```
num=${1:-3}
factor=1
times=0
while  [  $factor  -lt  13  ]
do
        let  times=num*factor
        echo "$num * $factor = $times"
        let  factor=factor+1
done
```

The script can be copied and executed as:

```
stimes
```

or

```
stimes  4
```

Infinite Loop

The following script, */scr/stimes1*, performs an infinite loop, which is ended by leaving the shell via the *exit* command:

```
num=${1:-3}
factor=1
times=0
while  true
do
        let  times=num*factor
        echo "$num * $factor = $times"
        let  factor=factor+1
        if  [  "$factor"  -gt  12  ]
        then
                exit
        fi
done
```

Here the test of True, a Unix/Linux built-in, always returns the variable $? as 0. The test of False always returns a non-zero value.

The script can be copied and executed as:

```
stimes1
```

or

```
stimes1  5
```

In the above example, the *exit* command exits from the loop and ends the shell script.

Exercise

1. Create a new script file *wcasename*, based on the file *casename,* and allow the user 3 attempts to enter the correct user name.

   Hint: use a *while* loop with a counter and make use of the *exit* command

for - Iteration

The command *for* may be used to cause a script to loop. The *for* loop executes a set number of times, based on a space separated set of values.

For example, the following script will place the values *a, b, c,* and *d* into the variable *i* and run the loop 4 times, once for each value.

```
# A shell script which runs a for loop
for  i  in  a  b  c  d
do
        echo $i
done
```

The space separated list is often constructed by use of the $(**…**) syntax with a valid command or set of commands.

This example will mail each user logged on:

```
for  user  in  $(who | cut  -f1  -d' ' | sort -u)
do
        echo  $user
done
```

Sequence of Numbers

The *for* loop executes a set number of times, very useful if needed for only 2 or 3 iterations. For a larger number, 20 perhaps, it could be coded as:

```
for  i  in  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

However, what if it was for 50 or 99 times!

The following **bash** shell examples generate a loop which runs 20 times:

```
# A shell script which runs a for loop 20 times
for  i  in {1..20}
do
        echo $i
done
```

or

```
for  i  in {20..1}
do
        echo $i
done
```

This example script uses a *for* loop to display a times table:

```
for  f  in  {1..12}
do
        echo  "${1:-3}  *  $f  =  $(let  x=${1:-3}*f;echo  $x)"
done
```

seq Command
The command *seq*, if available, may be used in both the **bash** and **korn** shells to generate a sequence of numbers. For example:

```
seq  1  20
```

To display the numbers on one line use:

```
seq  -s’  ‘ 1  20
```

This is used in the following example to generate a loop which runs 20 times:

```
# A shell script which runs a for loop 20 times
for  i  in $(seq  1  20)
do
        echo $i
done
```

This example will generate a loop based on the value of $1 but with a default value of 20:

```
# A shell script which runs a for loop many times
for  i  in $(seq  1  ${1:-20})
do
        echo $i
done
```

Exercise

1. Create a new script file *fcasename*, based on the script *casename,* and allow the user a number of attempts at entering a name.

   Hint: use a *for* loop and make use of the *exit* command.

shift - Moving Parameters

The command *shift* may be used in a loop to access the command line parameters passed to a shell script.

It moves the parameters to the left so that the 10th becomes the 9th, the 9th the 8th etc. and for each loop $1 changes.

For example, consider the following script, *shparam*:

```
# A shell script which uses the shift command

for  ans  in  $*
do
        echo  $*
        shift
done
```

The script can be tested by passing more than 10 parameters on the command line:

```
shparam  a  b  c  d  e  f  g  h  I  j  k  l  m  n
```

Exercise

1. Copy and test the script */scr/shparam* which is based on the above example.

while - Reading a File

Here, the *read* command is used to create two variables from the command line. By default, input is from the keyboard:

```
read  fn  ln
```

A *while* or *until* loop may also be used to read input from a file.

First display the file *people*:

```
cat  people
```

Now, redirect input from the file into the *while* loop, as in the script */scr/wread*:

```
while  read  dept  name  sal
do
        echo  “$name  $sal”
done  <  people
```

Exercise

1. Copy and test the script */scr/wread* which is based on the above example.

while - With a Pipe

If the input to the *while* loop is a command or a variable then a pipe must be used. In this example the contents of the file people are piped into the *while* command and an attempt is made to display the last name read:

```
cat people |

while  read  dept  name  sal
do
        echo -e "$name  $sal"
        last=$name
done
echo Last name is $last
```

However, this does not work as the variable *last* is created in the sub-shell forked by the *while* command and so is not available when the sub-shell ends.

To correct this use { }, which forces the *while* command to run in the current shell:

```
cat people |
{
 while  read  dept  name  sal
do
        echo -e "$name  $sal"
        last=$name
done
echo Last name is $last
}
```

Review Exercise

When running scripts, remember that they can be tested and **debugged** as follows:

```
bash  -xv  scriptname
```

1. *Using a for loop.*

   Create and test the script *fpcheck*, based on the file *pcheck,* and allow the user 3 attempts to enter a valid user name and password.

2. *Using a while loop that reads a file.*

   Create and test the script *rpcheck*, based on the file *pcheck*. Here the entered user name and password are tested against a tab separated file, *pfile*, which you should first create in your own user.

   This file contains valid user names and passwords in the form:

ken	spleen
alan	kidney
jenny	brain

   If you have time:

3. *Using a for loop with back quotes or $(…).*

   Amend the script *smess* so that if the user id input is *ALL*, it will write a message to all users logged on.

   Hint: use a *for* loop with the commands *who* and *cut* and *back quotes or $ (…)*.

Sub-scripts and Functions

Objectives

At the end of this section the delegate will be able to:

- call a sub-script and pass an exit status
- create local and global variables
- create and use a function
- list existing functions
- call a function from within a shell script

Sub-scripts

Scripts can be called from within other scripts. The following main script, *mainscr*, displays a LOCAL variable and calls a sub-script:

```
# A shell script which calls a sub script
user="steve"
echo  “Script  " $0 " User =  " $user
./subscr
```

The sub-script, *subscr*, also attempts to display the variable;

```
# A shell sub script which is called
echo  “Script  " $0 " User =  " $user
```

The two scripts can be tested by running the main script*:*

```
mainscr
```

Exercise

ˆ1 Copy the 2 scripts from the /scr directory and test them by running the script *mainscr*.

Export Command

Note that the sub script CANNOT display the variable *user* because it is LOCAL to the main script. To make the variable GLOBAL, the main script has to be amended and the variable *export*ed:

```
# A shell script which calls a sub script
user="steve"
export  user

echo  “Script  " $0 " User =  " $user
./subscr
```

The variable is now available because it has been exported to sub shells and the script will work and display the correct two messages:

```
mainscr
```

Exercise

ˆ1 Amend and test the script *mainscr* as in the example above.

Exit Status

The *exit* command may be used to exit from a script and pass back an exit status. If no value is passed then the default is the exit status of the last command.

In this example, the sub-script *subscr1* returns a value depending on whether or not the variable *$user* has a value. The -z option returns true if the value of the variable is NULL.

```
# A shell sub script which is called
echo  “Script  " $0 " User =  " $user

if  test  -z  "$user"
then
        exit  -1
else
        exit  0
fi
```

Note that because the subscript returns a value, this exit status may be tested in an *if* statement. Here the script *mainscr1* tests the exit status returned from the sub-script:

```
# A shell script which calls a sub script
user="$1"
export  user
echo  “Script  " $0 " User =  " $user

if  ./subscr
then
        echo "subscr returned 0"
else
        echo "subscr returned -1"
fi
```

Exercise

1̃ Copy the *mainscr1* and *subscr1* scripts from the /scr directory and test them by running the script *mainscr1*.

Functions

A function can be likened to a shell script stored in memory. However, a function like a variable, is only available within the shell in which it is created. This means that unless it is exported, a function created within a session or a shell script can only be called from within the session or script itself and not from within any other sessions or script files.

If created during a shell session, a function is lost when the session is ended. It can, however, be stored in one of the profile files executed on login.

Creating Functions

The following is an example of a simple function:

```
function  klist
{
        ls   -l  $1  |  more
}
```

To maintain compatibility with the Bourne shell, the following syntax may be used:

```
klist()
{       ls   -l  $1  |  more
}
```

The function can be entered at the **command prompt** as above and then tested as follows:

```
klist  /etc
```

Exercise

1. Create and test a function **at the command prompt** based on the above example.

Listing Functions

The command *type* can be used to list the function:

```
type  klist
```

The command *typeset* can be used to list available functions:

```
typeset  -f
```

To save these in a login file so that they are always available:

```
typeset  -f  >>  .bashrc
```

If saved in a file, the functions can be re-created by use of the dot command as follows:

```
.  .bashrc
```

The command *typeset* may also be used to export a function:

```
typeset  -fx  klist
```

A function may be removed by the command *unset* as follows:

```
unset  -f  klist
```

Example Function

Functions may also be used in place of a shell script. The following is an example of the *testname* script written as a function:

```
function  ftestname
{
        echo "Please enter your name: "
        read ans
        if  test  $ans  =  "Steve"
        then
                echo "Correct"
        else
                echo "Wrong"
        fi
}
```

Useful Functions

Many programming languages contain sets of useful functions, including ones to convert user input from upper to lower case and vice versa.

A *lower* function may be created as follows:

```
function  lower
{
        echo $1  |   tr  ‘[:upper:]’  ‘[:lower:]’
}
```

The above function may be tested thus:

```
lower  ABCD
```

Exercise

1̃ Create and test either a LOWER or UPPER function based on the above example.

Functions in Scripts

Many scripts contain calls to other sub scripts, using the relative pathname of the sub script. But if the main script is run from another user, the sub script will not be found.

This problem can be overcome by placing all scripts in a central directory such as */usr/local/bin* or for testing, in a local directory for the user such as *$HOME/bin*. The directory is then added to the PATH variable.

An alternative is to embed the sub script in the main script itself, replacing the call to the sub script with the actual script.

A better solution is to embed the sub script in the main script as a function and then to call the function within the main script.

However, a function is only available within the shell in which it is created. This means that a function created within a script can only be called from within the script and this in itself is a security feature.

The following is an example of a function embedded in a simple menu script:

```
# Simple menu script
function  ftestname
{
        echo "Please enter your name: "
        read ans
        if  test  $ans  =  "Steve"
        then
                echo "Correct"
        else
                echo "Wrong"
        fi
}
echo  "           Menu

        1: List users
        2: Test name

        Enter Selection: "

read ans
case  "$ans"  in
        1) who              ;;
        2) ftestname        ;;
        *)echo 'Enter valid selection'  ;;
esac
```

Exercise

1. Create a new script called *fname*, based on the script *casename*, which will use the LOWER or UPPER function to convert the input string.

   First embed the function in the actual script.

   Then, in the *case* statement, use the function to convert the input string and then test just for the correct case, lower or upper.

   Hint: use the $(…) syntax in the *case* statement.

Return Status

The *return* command may be used to exit from a function and pass on an exit status, for example:

```
function  ftestname
{
        echo "Please enter your name: "
        read ans
        if  test  $ans  =  "Steve"
        then
                echo "Correct"
                return  0
        else
                echo "Wrong"
                return  1
        fi
}
```

This return status may be tested in an *if* statement:

```
if  ftestname
then
        echo "ftestname returned 0"
fi
```

Review Exercise

1. *Calling a sub-script from a script.*

   Create a new script file *spcheck*, which again allows the user 3 attempts to enter the correct user name and password but this time call the script *pcheck* as a sub-script.

   In the main script *spcheck*, call the subscript *pcheck* in an IF statement and test the exit status returned.

   In the subscript *pcheck*, pass back an exit status of either 0 or -1.

2. *Calling a function from within a script.*

   Create and test a new script *spcheckf* based on the script *spcheck* but this time embed *pcheck* as a function rather than calling it as a script.