**Chapter 12**

**Shell Programming**

**1. Why shell programming?**

Even though there are various graphical interfaces available for Linux the shell still is a very neat tool. The shell is not just a collection of commands but a really good programming language. We can automate a lot of tasks with it, the shell is very good for system administration tasks, we can very quickly try out if our ideas work which makes it very useful for simple prototyping and it is very useful for small utilities that perform some relatively simple tasks where efficiency is less important than ease of configuration, maintenance and portability.

So let's see now how it works

**1.1 Creating a script**

There are a lot of different shells available for Linux but usually the bash (bourne again shell) is used for shell programming as it is available for free and is easy to use. So all the scripts we will write in this article use the bash (but will most of the time also run with its older sister, the bourne shell).

For writing our shell programs we use any kind of text editor, e.g. nedit, kedit, emacs, vi...as with other programming languages. The program must start with the following line (it must be the first line in the file)

**$ !/bin/sh**

The #! Characters tell the system that the first argument that follows on the line is the program to be used to execute this file. In this case /bin/sh is shell we use.

When we have written our script and saved it we have to make it executable to be able to use it. To make a script executable type

**$ chmod +x filename**

Then we can start our script by typing: ./filename

**1.2 Comments**

Comments in shell programming start with # and go until the end of the line. We really recommend to use comments. If we have comments and we don't use a certain script for some time we will still know immediately what it is doing and how it works.

**1.3 Variables**

As in other programming languages we can't live without variables. In shell programming all variables have the datatype string and we do not need to declare them. To assign a value to a variable we write:

**varname=value**

To get the value back we just put a dollar sign in front of the variable:

**$!/bin/sh**

Assign a value:

**$ a="hello world"**

Now print the content of "a":

**echo "A is:"**

**echo $a**

Type this lines into our text editor and save it e.g. as first. Then make the script executable by typing **chmod +x first** in the shell and then start it by typing ./first

The script will just print:

**A is:**

**hello world**

Sometimes it is possible to confuse variable names with the rest of the text:

**$num=2**

**echo "this is the $numnd"**

This will not print "this is the 2nd" but "this is the " because the shell searches for a variable called numnd which has no value. To tell the shell that we mean the variable num we have to use curly braces:

**$ num=2**

**echo "this is the ${num}nd"**

This prints what we want: this is the 2nd

There are a number of variables that are always automatically set. We will discuss them further down when we use them the first time.

If we need to handle mathematical expressions then we need to use programs such as expr. Besides the normal shell variables that are only valid within the shell program there are also environment variables. A variable preceded by the keyword export is an environment variable. We will not talk about them here any further since they are normally only used in login scripts.

**1.4. Exporting Variables**

By default, variables defined within a shell are local to it. Any process/command executed by that shell will not get access to these variables, unless they are exported to the environment.

**$ export VARNAME**

This export includes changes to existing variables that are in the path. for example, if we wish to modify the PATH environment variable we would:

**$ PATH=$PATH: additional\_directory**

**$ export PATH**

**1.5. Positional (Parameter) Variables**

When the shell script is run, any command line parameters to the shell are provided by a set of shell variables $0 $1 $2 $3 $4 $5 $6 $7 $8 and $9.

$0 (or ${0}) contains the name of the shell script itself, and depending on how the shell was called, may contain the full path to the command. If there are more than 9 parameters/arguments to the script, then they are accessed using the shift command which shifts all the parameters ($1=$2, $2=$3 etc) with $9 getting the next parameter in the line ($0 is unaffected).

$# is a variable that contains the number of command line parameters.

$\* and $@ can be used to get all the parameters in one go. Very useful for passing the complete command line parameter set to a child process

**Example**

**!/bin/sh**

A wrapper for the GNU C compiler that has the debug flag set

**$ gcc -g $\***

Note:

|  |  |
| --- | --- |
| $\* and $@ | are the same except when placing quotes around them |
| "$\*" | is the same as "$1 $2 $3..." |
| "$@" | is the same as "$1" "$2" "$3"... |
| $? | The exit status (error code) of the last command |
| $$ | The PID of the current command - Useful for creating uniquely named temp files (eg: tmp.$$) |
| $! | The process id of the last command started in the background (with &) |

Example:

Write these lines in a file

**Echo “First argument is $1”**

**Echo “Second argument is $2”**

**Echo “The Total number of arguments are $#”**

**Echo “The list of arguments are $\*”**

**Echo “The pid of the current process is $$”**

**Echo “The name of the file where we written this program is $0”**

**Echo “The exit status of last command $?”**

**Result:**

Execute the above program using some command line arguments like

**$ sh <filename> 1 2 3 4 5**

**1.6 Quoting**

Before passing any arguments to a program the shell tries to expand wildcards and variables. To expand means that the wildcard (e.g. \*) is replaced by the appropriate file names or that a variable is replaced by its value. To change this behaviour we can use quotes: Let's say we have a number of files in the current directory. Two of them are jpg-files, mail.jpg and tux.jpg.

**$ !/bin/sh**

**echo \*.jpg**

This will print "mail.jpg tux.jpg".

Quotes (single and double) will prevent this wildcard expansion:

**$ !/bin/sh**

**echo "\*.jpg"**

**echo '\*.jpg'**

This will print "\*.jpg" twice.

Single quotes are most strict. They prevent even variable expansion. Double quotes prevent wildcard expansion but allow variable expansion:

**$ !/bin/sh**

**echo $SHELL**

**echo "$SHELL"**

**echo '$SHELL'**

This will print:

**/bin/bash**

**/bin/bash**

**$SHELL**

Finally there is the possibility to take the special meaning of any single character away by preceeding it with a backslash:

**echo \\*.jpg**

**echo \$SHELL**

This will print:

**\*.jpg**

**$SHELL**

**2. shift: Shifts Parameters**

When a large number of parameters (;SPMgt; 9) are passed to the shell, shift can be used to read those parameters. If the number of parameters to be read is known, say three, a program similar to the following could be written:

**$!/bin/sh**

**echo The first parameter is $1.**

**shift**

**echo The second parameter is $1.**

**shift**

**echo The third parameter is $1.**

**exit 0**

**3. read: Reading Input from User**

The following short example shows how read can be used to get input from the user:

**#!/bin/sh**

**echo -e "Please enter your name: \c"**

**read NAME**

**echo "Your name is $NAME."**

**exit 0**

The \c means that the line feed will be suppressed, so that the prompt sits at the end of the line, not at the beginning of the following line.

Two more common controls available to the ***echo*** command are to use \n to add a line feed, and \t to add a tab. Multiple values may be read on a single line by using:

**$!/bin/sh**

**echo -e "Please enter two numbers: \c"**

**read NUM1 NUM2**

**echo The numbers entered are $NUM1 and $NUM2**

**exit 0**

This ensures that if two numbers are entered on a single line, they will be read within two variables. If three numbers were entered, the second variable (NUM2) would contain the last two numbers.

Assuming three numbers were the input of the above example, the first two numbers could be assigned to the first variable by entering them as

**num1 \num2 num3**

The backslash ( \) allows the blank space between num1 and num2 to be part of the variable (ordinarily, spaces are used as **field seperators.**

**4. Conditional Statements**

**4.1. if Statements**

The core condi3tional concept of wrapping a block of statements that is only to be processed ***if*** some condition is met.

Shells also support ***else*** and the combined ***elif*** else-if condition, 3 basic layouts for if statements are shown below. Note the use of the then and keyword to separate the condition commands from internal command block, and the ***fi*** keyword to mark the end of the block.

**First Model:**

**if condition-command**

**then**

**command1**

**command2**

**...**

**fi**

**Second Model:**

**if condition-command**

**then**

**commandA1**

**commandA2**

**...**

**else**

**commandB1**

**commandB2**

**...**

**fi**

**Third model:**

**if condition-command-A**

**then**

**commandA1**

**commandA2**

**...**

**elif condition-command-B**

**then**

**commandB1**

**commandB2**

**...**

**else**

**commandC1**

**commandC2**

**...**

**fi**

The commands inside the blocks are used the same as any other command within the system, and it is possible to nest other conditions statements inside those blocks. For conciseness, many people will use the semicolon (;) command separating character to allow the then keyword to be placed on the same line as the if and the condition-command. (Note that no semicolon is needed after the then keyword)

The condition-command is a any command, and the if statement is evaluated based on the success (exit status) of that command.

**Example:**

**if ls -al ; then**

**echo "Directory was successfully listed"**

**else**

**echo "Failed to list directory"**

**fi**

**4.2. The test or [ ] Command**

There are other types of tests beyond running a simple command and checking its exit status. The shells support a number for "test" commands that can be used to perform more useful tests.

The test command has two formats:

**test expression**

**or**

**[ expression ]**

Note: for the [ ] version there must be spaces between the brackets and the expression.

The test command/statement will evaluate the expression and return a status of success if the expression is true. The following tests can be performed:

**4.2.1 File Tests**

|  |  |
| --- | --- |
| -r file | true if file is readable (by current user) |
| -w file | true if file is writable (by current user) |
| -c file | true if file is executable (by current user) |
| -f file | true if file is an ordinary file |
| -d file | true if file is a directory |
| -s file | true if the length of file is > 0 |
| -t | true if standard output is associated with a terminal |
| -t fd | true if file descriptor 'fd' is associated with a terminal |

An example would be where a program needs to output something to a file, but first checks that the file exists:

**#!/bin/sh**

**echo “Enter the filename”**

**read file1**

**if [ ! -s file1 ] or if test ! –s file1**

**then**

**echo "file1 is empty or does not exist."**

**ls -l > file1**

**exit**

**else**

**echo "File file1 already exists."**

**fi**

**exit 0**

**4.2.2 String Tests**

|  |  |
| --- | --- |
| -z str | true if the length of str is zero |
| -n str | true if the length of str is non zero |
| str | true if the str is not empty |
| str1 = str2 | true if str1 and str2 are the same |
| str1 != str2 | true if str1 and str2 are different |

An example would checks that the given strings are same or not:

**#!/bin/sh**

**echo “Enter the strings(string1,string2)”**

**read str1**

**read str2**

**if test str1 = str2**

**then**

**echo "Both Strings are equal"**

**exit**

**else**

**echo "Given strings are not equal"**

**fi**

**exit 0**

**4.2.3 Numeric Tests**

|  |  |
| --- | --- |
| int1 -eq int2 | true if int1 = int2 |
| int1 -ne int2 | true if int1 != int2 |
| int1 -lt int2 | true if int1 < int2 |
| int1 -le int2 | true if int1 <= int2 |
| int1 -gt int2 | true if int1 > int2 |
| int1 -ge int2 | true if int1 >= int2 |

**Example:**

**$!/bin/sh**

**if test $# -le 5**

**then**

**echo Less than or equal to five parameters.**

**else**

**echo More than 5 parameters.**

**fi**

**exit 0**

**4.2.4 Combining Operators**

* ! NOT - Invert the result of the next expression
* -a AND - Only true if both prev and next expr are true
* -o OR - True if either prev or next expr is true
* ( ) parentheses for grouping expressions

All parameters to test must be separated by spaces, including parentheses, operators and strings/integers. Note, that test operates on strings (or strings containing integer numbers) It cannot process commands directly, we must use the back-quotes to perform command substitution.

**if [ "$HOME" = `pwd` ] ; then echo "I am home" ; fi**

Also double quoting of variable is important within a test statement, as an undefined variable will resolve to nothing which will not be correctly processed by the shell.

For example, if $HOME is undefined

**if [ $HOME = `pwd` ] ; then...**

will expand to be:

**if [ = /export/home/me ] ; then...**

Which is invalid.

**4.3. case Statements**

When there are several different sets of operations to be performed based on different values of a single string, it can get quite messy using a long string of if, elif, elif, elif, else statements to perform the operations. The case command allows a convenient structured method of performing flow control through one of multiple blocks using pattern matching of strings.

**$ case string in**

**pattern1 ) commands;;**

**pattern2 ) commands ;;**

**pattern3 ) commands ;;**

**...**

**\*) commands ;; # default case**

**esac**

In addition, multiple pattern options can be specified for a single case using pattern1a | [I pattern1b] style blocks.

When specifying patterns,

* \* matches any string of zero or more characters
* ? matches any single character
* [ ] matches any single character within the set
* [! ] matches any single character not in the set

These can be combined together to form more advanced patterns:

[Yy]\* Matches any string starting with an upper or lower case y.

Use quotes or escape/back-slash the special characters if we wish to pattern match them specifically.

**$!/bin/sh**

# an example with the case statement, which reads a command from the user and processes it

**echo "Enter your command (who, list, or cal)"**

**read command**

**case "$command" in**

**who)**

**echo "Running who..."**

**who**

**;;**

**list)**

**echo "Running ls..."**

**ls**

**;;**

**cal)**

**echo "Running cal..."**

**cal**

**;;**

**\*)**

**echo "Bad command, your choices are: who, list, or cal"**

**;;**

**esac**

**exit 0**

**4.4. while Statements .**

Similar to the basic if statement, except the block of commands is repeatedly executed as long as the condition is met.

**while condition-command**

**do**

**command1**

**command2**

**...**

**done**

As with if statements, a semicolon (;) can be used to remove include the do keyword on the same line as the while condition-command statement.

The example below loops over two statements as long as the variable i is less than or equal to ten. Store the following in a file named while1.sh and execute it

**$!/bin/sh**

Illustrates implementing a counter with a while loop

Notice how we increment the counter with ***expr*** in backquotes

**i="1"**

**while [ $i -le 10 ]**

**do**

**echo "i is $i"**

**i=`expr $i + 1`**

**done**

**4.5 until loop**

Execute statements as long as a condition is false

**until grep "sort" dbase\_log > /dev/null**

**do**

**sleep 10**

**done**

**echo "Database has been sorted"**

Example executes until grep is unsuccessful

**4.6 for Statement**

The ***for*** statement is used to iterate/loop through a predefined list of values, setting a shell variable to the current value each time through the loop. Unlike many other languages shell for loops operate on word lists, not integers with start and stop values.

**for VAR in wordlist**

**do**

**commands**

**...**

**done**

**Example:**

**#!/bin/sh**

**for i in Claude Paul Wayne Roger Tom**

**do**

**echo The name is $i.**

**done**

**exit 0**

Within the shell, parameters are read as $1 for the first parameter, $2 for the second parameter, $3 for the third parameter, and so on. $\* is the entire list of parameters. If the ``in list " is omitted, the list taken is the list of parameters passed to the shell on the command line.

Note: To excite above said example programs we have to compile them in a file and excite that file. Like

**$ sh <filename>**

**5. Debugging**

The simplest debugging help is of course the command echo. We can use it to print specific variables around the place where we suspect the mistake. This is probably what most shell programmers use 80% of the time to track down a mistake. The advantage of a shell script is that it does not require any re-compilation and inserting an "echo" statement is done very quickly.

The shell has a real debug mode as well. If there is a mistake in our script "strangescript" then we can debug it like this:

**$ sh -x strangescript**

This will execute the script and show all the statements that get executed with the variables and wildcards already expanded.

The shell also has a mode to check for syntax errors without actually executing the program. To use this run:

**$ sh -n our\_script**

If this returns nothing then our program is free of syntax errors.