

Introduction

The shell provides a simple user interface to the UNIX system providing the user with a command line interpreter. Shells also provide a powerful programming language with which we can create shell programs (sometimes called shell scripts) to create useful software tools that can be used to control or administer a UNIX system. A shell program is simply a file containing a set of UNIX commands that are to be executed sequentially. The file needs to have executed permissions set on it so that it can be executed just by typing in the name of the file at the command prompt.

The shell provides much more than simply 'batch' processing of a list of UNIX commands. It has many features of a high-level programming language, such as:

- Variables for storing data
- Decision-making controls (the **if** and **case** statements)
- Looping controls (the **for,while** and **until** loops)
- Function calls for program modularity

The best way to study shell programming is to look at as many examples as possible. Each of the following examples are written in 3 parts:

- The shell program. (The lines are numbered only to aid with the explanation, do not type them into the program)
- The output generated by executing the shell program
- An explanation of the program

Type each of these example programs into a file and run them, make sure that you understand what each program does and why.

To execute a shell program held in a file named **shelldemo1** do the following

**\$ chmod +x shelldemo1**  
(This puts execute permission on the file - a topic we will look at in later sessions)  
**\$ ./shelldemo1**  
(This runs or executes the program. The dot represents your current directory)

**Demo Program 1 - Demonstrates the use of comments, user-defined variables and echo.**

1. #!/bin/sh
2. #Filename: shelldemo1 Author: RS
3. #Define variables
4. <b>name</b> =John
5. <b>car</b> ="Ford Escort"
6. <b>age</b> =21
7. #Display the contents of the variables
8. echo "My name is <b>\$name</b> "
9. echo "I am <b>\$age</b> years old and \c"
10. echo "I drive a <b>\$car</b> ."

Program output

My name is John  
I am 21 years old and I drive a Ford Escort.

Demo Program 2 - Demonstrates how the output of Unix commands can be stored in user-defined variables

1. #!/bin/sh
2. #Filename: shelldemo2   Author: RS
3. #Define variables
4. <b>todaysdate</b> =`date`
5. <b>myworkingdirectory</b> =`pwd`
6. #Display the contents of the variables
7. echo "The date is <b>\$todaysdate</b> "
8. echo "My current working directory is <b>\$myworkingdirectory</b> "

Program output

The date is Fri Aug 17 14:30:58 BST 2001  
My present working directory is /home/martin/shelldemos

---

Demo Program 3 - Demonstrating how the read command is used to get input from the user via the keyboard.

1. #!/bin/sh
2. #Filename: shelldemo3   Author: RS
3. echo "Please enter your first name: \c"
4. read <b>firstname</b>
5. echo "Please enter your surname: \c"
6. read <b>surname</b>
7. echo "Please enter your date of birth: (dd/mm/yyyy): \c"
8. read <b>dateofbirth</b>
9. echo "Welcome <b>\$firstname</b> <b>\$surname</b> , your date of birth is on record as <b>\$dateofbirth</b> ."

Program output (user input is shown in *bold italics*)

Please enter your first name: ***Joe***  
Please enter your surname: ***Bloggs***  
Please enter your date of birth (dd/mm/yyyy): ***01/04/1980***  
Welcome Joe Bloggs, your date of birth is on record as 01/04/1980

---

Demo Program 4 - Demonstrating how user input can be obtained from the command line as command line arguments.

1. #!/bin/sh
2. #Filename: shelldemo4   Author: RS
3. echo "This program has obtained its input from the command line"
4. echo "Welcome <b>\$1</b> <b>\$2</b> , your date of birth is on record as <b>\$3</b> ."

Program output

***This program is executed by entering the command:***  
***./shelldemo4 Joe Bloggs 01/04/1980***

Welcome Joe Bloggs, your date of birth is on record as 01/04/1980.

---

Explanation

This program is executed by typing in the shell program name followed by 3 pieces of information (program arguments).  
The positions of the words on the command line are identified by the following special variables (here is the full list):  
\$0 The command name  
\$1 The first argument (Joe in this example)  
\$2 The second argument (Bloggs in this example)  
\$3 The third argument (01/04/1980 in this example)  
.  
.  
\$9 The ninth argument  
\$# The number of arguments  
\$\* A space separated list of all the arguments.

---

Demo Program 5 - Demonstrating how decisions are made using the *if...then* statement (testing and branching).

1.	#!/bin/sh
2.	#Filename: shelldemo5 Author: RS
3.	clear
4.	echo "Would you like to see a joke (y/n)? \c"
5.	read reply
6.	<b>if</b> [ "\$reply" = "y" ]
7.	<b>then</b>
8.	echo "Question: How many surrealists does it take to change a light bulb?"
9.	echo "Answer: Fish."
10.	<b>fi</b>
11.	echo "\n\nHave a nice day."

Program output (examples of both yes and no user responses are shown)

Would you like to see a joke (y/n)? *y*  
Question: How many surrealists does it take to change a light bulb?  
Answer: Fish

Have a nice day.

Would you like to see a joke (y/n)? *n*

Have a nice day

Line 6 - The start of the if statement. Notice the test is in square brackets with a space either side of them and that there is a space either side of the equals sign.

Demo Program 6 - Demonstrating how decisions are made using the *if...then...else* statement (testing and branching).

1.	#!/bin/sh
2.	#Filename: shelldemo6 Author: RS
3.	clear
4.	echo "Would you like to see a joke (y/n)? \c"
5.	read reply
6.	<b>if</b> [ "\$reply" = "y" ]
7.	<b>then</b>
8.	echo "Question: How many surrealists does it take to change a light bulb?"
9.	echo "Answer: Fish."
10.	<b>else</b>
11.	echo "Not in the mood for jokes? Never mind perhaps another day."
12.	<b>fi</b>
13.	echo "\n\nHave a nice day."

Program output (examples of both yes and no user responses are shown)

Would you like to see a joke (y/n)? *y*  
Question: How many surrealists does it take to change a light bulb?  
Answer: Fish.

Have a nice day.

Would you like to see a joke (y/n)? *n*  
Not in the mood for jokes? Never mind perhaps another day.

Have a nice day

Demo Program 7 - Demonstrating how decisions are made using nested *if...then...else* statements (testing and branching).

1.	#!/bin/sh
2.	#Filename: shelldemo7 Author: RS

3.	echo "UNIX COMMAND SELECTOR"
4.	echo "1. Show date"
5.	echo "2. Show hostname"
6.	echo "3. Show this month's calendar"
7.	echo "Please make your selection (1,2,3) \c"
8.	read menunumber
9.	if [ \$menunumber -eq 1 ]
10.	then
11.	date
12.	else if [ \$menunumber -eq 2 ]
13.	then
14.	hostname
15.	else if [ \$menunumber -eq 3 ]
16.	then
17.	cal
18.	else
19.	echo "INVALID CHOICE! \07\07"
20.	fi
21.	fi
22.	fi
23.	echo "\nThank you for using the Unix command selector."

**Program output (The example is for the user input of 1)**

```
UNIX COMMAND SELECTOR
1. Show date
2. Show hostname
3. Show this month's calendar
Please make your selection (1,2,3) 1
Tues Oct 23 17:32:45 BST 2001

Thank you for using the Unix command selector.
```

**Explanation**

Lines 3-7 Display a title, menu and prompt the user to select the number of a menu option.  
Line 8 - The user's response is read into a user defined variable named *menunumber*.  
Lines 9-22 A series of nested if...then...else statements each performing a test for one of the possible menu option numbers.  
The 3 possible Unix commands are: 1. date 2. hostname 3. cal  
Notice how **-eq** is used to test for equality between two numbers.  
Notice how \07\07 are used to get the computer to beep twice.  
Line 23 Displays a final ending message.

Nested **if...then...else** structures always look complicated as there is a lot of coding to contend with. A simpler way of writing the nested **if...then...else** parts is shown below using the **if...then...elif** with one final **fi**. The word **elif** is a contraction of the words **else if**.

```
if [ $menunumber -eq 1 ]
then
    date
elif [ $menunumber -eq 2 ]
then
    hostname
elif [ $menunumber -eq 3 ]
then
    cal
else
    echo "INVALID CHOICE! \07\07"
fi
```

This may be a little better to deal with but nested **if...then...elif's** are still a handful. Let's look at a neater solution to the same problem using the **case...esac** program structure shown in the next example.

**Demo Program 8 - Demonstrating how decisions are made using the *case...esac* statement (testing and branching).**

1.	echo "UNIX COMMAND SELECTOR"
2.	echo "1. Show date"
3.	echo "2. Show hostname"
4.	echo "3. Show this month's calendar"
5.	echo "Please make your selection (1,2,3) \c"
6.	read menunumber
7.	<b>case</b> \$menunumber <b>in</b>
8.	<b>1)</b> date;;
9.	<b>2)</b> hostname;;
10.	<b>3)</b> cal;;
11.	<b>*)</b> echo "INVALID CHOICE! \07\07";;
12.	<b>esac</b>
13.	echo "\nThank you for using the Unix command selector."

**Program output (The example is for the user input of 1)**

UNIX COMMAND SELECTOR  
1. Show date  
2. Show hostname  
3. Show this month's calendar  
Please make your selection (1,2,3) *1*  
Tues Oct 23 17:32:45 BST 2001

Thank you for using the Unix command selector.

**Explanation**

Line 7 - The first line of the *case...esac* statement checks to see the value held in the variable named menunumber.  
Lines 8-11 give possible branch conditions depending upon the value held in the variable menunumber. The content of the variable is checked against the value 1, 2 or 3 or anything else (represented by the asterisk). Note the double semicolon at the end of each test condition. (Isn't program syntax wonderful!)  
Line 12 - The word **esac** is **case** spelt backwards. It identifies the end of the case statement.

**Demo Program 9 - Demonstrating how looping is achieved using the *for* statement.**

1.	echo "Demonstration of looping using the for loop and a list of car names"
2.	<b>for</b> car <b>in</b> ford vauxhall rover toyota mazda subaru
3.	<b>do</b>
4.	echo \$car
5.	<b>done</b>
6.	echo "\nEnd of demonstration program."

**Program output**

Demonstration of looping using the for loop.  
ford  
vauxhall  
rover  
toyota  
mazda  
subaru

End of demonstration program.

**Explanation**

Line 2-5 The general format of a for loop is:  
*for* variable *in* list\_of\_items  
*do*

commandA  
commandB  
commandC  
*done*

The keywords are: *for*, *in*, *do* and *done*. In this example *car* is a user-defined variable and the list of data items are: *ford vauxhall rover toyotamazda subaru* all separated with spaces. The example only has one command belonging to the for loop which resides between the keywords *do* and *done*. The content of the variable named *car* has a different value for each pass through the loop.

Demo Program 10 - Demonstrating how looping is achieved using the *for* statement.

1. echo "Demonstration of looping using the for loop and a list of filenames generated by the ls command"
2. <b>for</b> myfile <b>in</b> `ls`
3. <b>do</b>
4.     cat \$myfile
5. <b>done</b>

Program output

The output seen on the screen would be the contents of all the files held in the current directory.

Explanation

Line 2 - The loop variable is named myfile. The list of data items is generated by the Unix command *ls*. Each pass through the loop then displays the contents of a file by using the Unix *cat* command. Note the use of backquotes around the *ls* command.

Demo Program 11 - Demonstrating how looping is achieved using the *for* statement.

1. echo "Demonstration of looping using the for loop and a list of filenames held in a text file named myfilelist"
2. <b>for</b> myfile <b>in</b> `cat myfilelist`
3. <b>do</b>
4.     cat \$myfile
5. <b>done</b>

Program output

The output seen on the screen would be the contents of all the files that have their names listed in the textfile named *myfilelist*.

Explanation

Line 4 - The loop variable is named myfile. The list of data items is held in a text file named myfilelist. The list of items is generated by the Unix command *cat myfilelist*. Each pass through the loop then displays the contents of a file that has its name listed in the text file named myfilelist. Note the use of backquotes around the *cat myfilelist* command.

Demo Program 12 - Demonstrating how looping is achieved using the *for* statement.

1. echo "Demonstration of looping using the for loop and a list of arguments supplied at the command line"
2. echo "Program invoked by the command:   ./shelldemo12   filename1 filename2   filename3"
3. <b>for</b> myfile <b>in</b> \$*
4. <b>do</b>
5.     cat \$myfile
6. <b>done</b>

Program output

The output seen on the screen would be the contents of all the files listed on the command line when the shell program is executed.

Explanation

Line 3 - The loop variable is named myfile. The list of data items is represented by the special variable name \$\* (see explanation of demo program 4 for full list of special variables).

Demo Program 13 - Demonstration of 'while' loop

You can use ((expression)) syntax to test arithmetic evaluation (condition). To replace while loop condition **while** [ \$n -le 5 ] with **while** (( num <= 10 )) to improve code readability:

```
#!/bin/bash
n=1
while (( $n <= 5 ))
do
    echo "Welcome $n times."
    n=$(( n+1 ))
done
```

Program outputs:

```
Welcome 1 times.
Welcome 2 times.
Welcome 3 times.
Welcome 4 times.
Welcome 5 times.
```

Globally search for Regular Expression and Print (GREP)

The grep command globally searches for regular expressions in files and prints all lines that contain the expression. The egrep and fgrep commands are variants of grep. The egrep is extended version of grep supporting more metacharacters (we will see them in a table very soon). The fgrep command called fixed grep or sometimes called fast grep searches for fixed strings (see man page for details). We will only discuss grep as it is more commonly used. The grep command searches for a pattern of characters in text supplied to it from:

- 1. a text file or number of files (command format is: **grep pattern filename** )
- 2. the output of another UNIX command that is piped to the grep command (command format: **unixcommand | grep pattern** )

The grep command sends its output to the screen and does not change or affect the file(s) being accessed.

Method 1: In conjunction with a text file

- The commands are entered at the Unix shell prompt.
- Single quotes are used if the search string contains metacharacters or spaces.
- The examples are based on the following file named **datafile**:

datafile
Abel Adams 80 90
Brian Brown 50 40
Cath Cookson 30 55
Dave Davidson 40 60
David Smith 60 40
Eric Erikson 60 80

Try out each of the following to prove that it works

	Metachar	Meaning	Example	Explanation
1			grep Davidson datafile	Print any line from the file named test file that contains the text " Davidson"
2	^	Beginning of line anchor	grep '^A' datafile	Print any line beginning with A.
3	\$	End of line anchor	grep '5\$' datafile	Prints any line ending with the character 5.
4	.	Matches one character	grep 'ks..' datafile	Prints any line containing ks followed by any 2 single characters.
5	*	Matches zero or more characters	grep 'oo*' datafile	Print any line containing a letter o followed by zero or more consecutive letter os.

Use of grep command line options ( Try out any of the following that don't appears obvious to you )

1	grep -n Adams datafile	Print any line from the file that contains Adam and show line number.
2	grep -n '^Cath' datafile	Print any line from the file that begins with Cath and show line number.
3	grep -i 'brian' datafile	The -i option turns off case sensitivity. It does not matter if the expression contains any combination of upper or lowercase letters.
4	grep -v 'Eric Erikson' datafile	Prints all lines not containing the pattern Eric Erikson. This is useful when removing someones record from a file... look at this: <b>grep -v 'Eric Erikson' datafile &gt; tempfile</b> <b>mv tempfile datafile</b>
5	grep -c 'Dav' datafile	The -c option causes grep to print the number of lines (not occurrences) where the pattern was found.

Method 2: In conjunction with the information piped to it from another UNIX Command

Try out each of the following and ensure that they make sense to you.

Unix command	Description
ls -l	List the current directory in long format. The output automatically goes to the screen.
ls -l   grep '^d'	List the current directory in long format and pipe the output of this command to the grep command which lists all lines beginning with the lowercase letter d. This example identifies the subdirectories that are listed by the ls -l command.
ls -l   grep '^_'	List the current directory in long format and pipe the output of this command to the grep command which lists all lines beginning with a dash. This example identifies the ordinary files that are listed by the ls -l command.