# Bourne/Bash:
# Shell Programming Introduction

**By Rick Dearman**

---

Sooner or later every UNIX user has a use for a shell script. You may just want to do a repetitive task easier, or you may want to add a bit more kick to an existing program. An easy way to accomplish this is to use a shell script. One of the first shell scripts I wanted was something that would change a directory full of files which were all in capital letters to lowercase. I did it with this script:

LCem.sh

```
1       #!/bin/sh
2
3       DIR=$1
4
5       for a in `ls $DIR`
6       do
7            fname=`echo $a | tr A-Z a-z`
8           mv $DIR/$a $DIR/$fname
9       done;
10       exit 0
11        #this script will output and error if the file is already lowercase, and
assumes argument is a directory
```

Line one tells the computer which shell to use, in this case it is "sh" the bourne shell ( or this may be a link to the bash shell ). The combination of the two symbols #! are special to the shell and indicates what shell will run this script. It IS NOT IGNORED like other comment lines. Line 3 sets a variable called DIR to equal the first argument of the input. (Arguments start at $0, which is the name of the shell script or in this case LCem.sh ).

In line 5 we enter a control loop. In this case it is a for loop. Translated into english this line means for every entry "a" that I get back from the command `ls $DIR` I want to do something. The shell will replace the variable name $DIR to whatever was typed on the command line for you. Line 6 starts the loop.

Now in line seven we make use of the UNIX utilities available , `echo` and `tr`. So what we are doing is echoing whatever the current value of $a is and piping it into tr

which is short for translate. In this case we are translating uppercase to lowercase, and setting a new variable called fname to the result.

In line eight we move the file $DIR/$a, whatever it may be to $DIR/$fname. Line nine tells the shell to go back and do all the other $a variables until it is done. And finally line 10 we exit the script with an error code of zero. Line eleven is a comment.

This script wouldn't have been needed to change one or two file names, but because I needed to change a couple of hundred it saved me lots of typing. To get this to run on your machine you would have to chmod the file to be executable. Like this `chmod +x LCem.sh` . Or you could evoke the shell command directly and give it the name of your script like this `sh LCem.sh`. Using the comment and exclamation mark combination would tell the kernel what shell to evoke and is the normal way to do things. But remember if you use the #! then the file itself needs to have execution permissions.

It is only eleven lines but it shows us a lot about shell scripting. We have learned how to get the computer to run the script using the #! combination. This combination of a comment mark and a bang operator, or as some people call it an exclamation mark, is used to start a shell script without having to evoke the shell first. We learned that a # is how we can write a comment into our script and have them ignored when the script is processed. We learned how to pass arguments to the script to get input from the user, and we know how to set a variable. We have glanced at one of the many control structures we can use to control the functionality of a script.

Don't worry if you didn't really get all of that. We shall now move on to explaining some of the most common decision making / control structures. The first one we want to look at is the `if` statement. In every programming language we want to be able to change the flow of the program based on various conditions. For example if a file is in this directory do one thing. If it isn't do something else. The syntax for the if command is:

```
if expression then
   commands
fi
```

So if the expression is true the statements inside the if block are executed. Lets look at a simple example of the if statement.

WhoMe.sh

```
1   #!/bin/sh
2
3   # set the variable ME to the first argument after the command.
4   ME=$1
5
6   # grep through the passwd file discarding the output and see if $ME is in the file
7   if  grep $ME /etc/passwd > /dev/null
8   then
9   # if $ME is in the file out put the following line
10        echo "You are a user"
11    fi
```

Notice the extensive use of comments on lines 3, 6, and 9. You should try to comment you scripts as much as possible because someone else may need to look at it later. In six months you may not remember what you were doing, so you might need the comments as well.

Using the if statement we can now correct some of the errors which would occur in the lowercasing script. In LCem.sh the script will hang if the user doesn't input a directory as an argument. To check for an empty string, we would use the following syntax:

```
if [ ! $1 ]
```

This means if not $1. The two new things here are the use of the bang operator, or exclamation mark as the symbol for NOT.  So lets add this new knowledge to our program.
#!/bin/sh

```
1   if [ ! $1 ]
2   then
3   echo "Usage: `basename $0` directory_name"
4    exit 1
5   fi
6
7   DIR=$1
8
9     for a in `ls $DIR`
10   do
11          fname=`echo $a | tr A-Z a-z`
```

```
12     mv $DIR/$a $DIR/$fname
13    done;
```

Now if the user types in the command but not the directory then the script will exit with a message about the proper way to use it, and an error code of one.

But what if we really did want to change the name of a single file? We have already got this command wouldn't it be nice if it could cope. If we want to do that then we need to be able to test if the argument is a file or directory. Here is a list of the file test operators.

| **Parameter** | **Test** |
|---|---|
| -b file | True is file is a block device |
| -c file | True if file is a character special file |
| -d file | True if the file is a directory |
| -f file | True if file is a ordinary file |
| -r file | True if file is readable by process |
| -w file | True if file is writeable by process |
| -x file | True if file is executable |

There are more operators but these are the most commonly used ones. Now we can test to see if the user of our script has input a directory or a file. so lets modify the program a bit more.

```
1    #!/bin/sh
2
3    if [ ! $1 ]
4    then
5     echo "Usage: `basename $0` directory_name"
6     exit 1
7    fi
8
9    if [ -d $1 ]
10    then
11     DIR="/$1"
12    fi
13
14    if [ -f $1 ]
15    then
```

```
16    DIR=""
17   fi
18
19   for a in `ls $DIR`
20   do
21       fname=`echo $a | tr A-Z a-z`
22       mv $DIR$a $DIR$fname
23   done;
```

We inserted lines nine through seventeen to do our file/directory checks. If it is a directory we set DIR to equal "/$1" if not we set it blank. Notice we now put the directory slash in with the DIR variable and we've modified line 22 so that there is no slash between $DIR and $a. This way the paths are correct.

We still have a few problems with our script. One of them is that if the file which is getting moved already exists then the scripts outputs an error. What we want to do is check the file name before we attempt to move it. Another thing is what if someone puts in more than two arguments? We'll modify our script to accept more than one path or filename.

The first problem is easily corrected by using a simple string test and an if statement like we have use earlier. The second problem is slightly more difficult in that we need to know how many arguments the user has input. To discover this we'll use a special shell variable which is already supplied for us. It is the $# variable, this holds the number of arguments present on the command line. Now what we want to do is loop through the arguments until we reach the end. This time we'll use the While loop to do our work. Finally we shall need to know how to compare integer values, this is because we want to check the number of time we have gone through the loop to the number of arguments. There are special test options for evaluating integers, they are as follows

| Test | Action |
|---|---|
| int1 -eq int2 | True if integer one is equal to integer two |
| int1 -ge int2 | True if integer one is greater than or equal to integer two |
| int1 -gt int2 | True if integer one is greater than integer two |
| int1 -le int2 | True if integer one is less than or equal to integer two |
| int1 -lt int2 | True if interger one is less then interger two. |
| int1 -ne int2 | True if integer one is not equal to integer two |

Using this new knowledge we'll modify our program.

```
1   #!/bin/sh
2
3   if [ ! $1 ]
4   then
5       echo "Usage: `basename $0` directory_name"
6    exit 1
7   fi
8
9   while [ $# -ne 0  ]
10   do
11              if [ -d $1 ]
12              then
13                  DIR="/$1"
14               fi
15
16               if [ -f $1 ]
17              then
18                  DIR=""
19               fi
20
21               for a in `ls $DIR`
22               do
23                     fname=`echo $a | tr A-Z a-z`
24               if [ $fname != $a ]
25                then
26                        mv $DIR$a $DIR$fname
27               fi
28              done;
29
30              shift
31   done
```

What we've done here is to insert a while loop on line 9 which checks to see if the arguments listing is equal to zero. This may seem like we just created an infinite loop but the command on line 30 the shift saves us. You see the shift command basically discards the command nearest the command name. (LCem.sh) and replaces it with the one to the right. This loop will succeed in discarding all the arguments eventually and then will equal zero and exit our loop.

And finally note the if statement on line 24, this checks to see if the file name is already lowercase and if so ignores it.

I hope you have enjoyed this brief introduction to Bourne / Bash programming. I would encourage you to try some of these examples for yourself. In fact if you want you could make this script much better by using a switch like -l to lowercase and -u to uppercase and modifying the script to handle it.

I take full responsibility for any errors or mistakes in the above documentation. Please send any comments or questions to rick@ricken.demon.co.uk

REFERENCES:

The UNIX programming environment
by Brian W. Kernighan & Rob Pike
Published by Prentice Hall

Inside UNIX
Published by New Riders