steve-parker.org

Unix & Linux Shell Scripting Tutorial

Steve Parker

Unix & Linux
Shell Scripting Tutorial

(c) 2000 – 2014 Steve Parker

A Bourne Shell Programming/Scripting Tutorial for learning about using the Unix shell.

- Join in on Facebook, at http://facebook.com/shellscript
- Check out the *nixShell Blog at http://nixshell.wordpress.com/
- See this book's Kindle page at http://amzn.com/dp/B00C2EGNSA

Version:1.2h

Table of Contents

#!/ch/1

1. Introduction

Purpose Of This Tutorial

This tutorial is written to help people understand some of the basics of shell script programming, and hopefully to introduce some of the possibilities of simple but powerful programming available under the bourne shell. As such, it has been written as a basis for one-on-one or group tutorials and exercises, and as a reference for subsequent use.

A Brief History of sh

Steve Bourne wrote the Bourne shell which appeared in the Seventh Edition Bell Labs Research version of Unix. Many other shells have been written; this particular tutorial concentrates on the Bourne and the Bourne Again shells. Other shells include the Korn Shell (ksh), the C Shell (csh), and variations such as tcsh. This tutorial does *not* cover those shells. Maybe a future version will cover ksh; I do not intend to write a tutorial for csh, as csh programming is considered harmful[1].

Audience

This tutorial assumes some prior experience; namely:

- Use of an interactive Unix shell
- Minimal programming knowledge - use of variables, functions,

1 See http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/

is useful background knowledge
- Understanding of how Unix commands are structured, and competence in using some of the more common ones.
- Programmers of perl, python, C, Pascal, or any programming language (even BASIC) who can maybe read shell scripts, but don't feel they understand exactly how they work.

Typographical Conventions Used in This Tutorial

Significant words will be written in *italics* when mentioned for the first time. Code segments and script output will be displayed as monospaced text. Command-line entries will be preceded by the Dollar sign ($). If your prompt is different, enter the command:

```
PS1="$ " ; export PS1
```

Then your interactions should match the examples given (such as $ ./my-script.sh below). Script output (such as "Hello World" below) is displayed at the start of the line.

Text that is to be entered by you, is shown in **bold text**.

```
$ echo '#!/bin/sh' > my-script.sh
$ echo 'echo Hello World' >> my-script.sh
$ chmod 755 my-script.sh
$ ./my-script.sh
Hello World
$
```

Entire scripts will be surrounded by thick horizontal rules and include a reference where available to where the plain text of the script may be downloaded (from http://steve-parker.org/sh/eg):

first.sh[2]

```
#!/bin/sh
# This is a comment!
echo Hello World        # This is a comment, too!
```

Note that to make a file executable, you must set the eXecutable bit, and for a shell script, the Readable bit must also be set:

```
$ chmod a+rx first.sh
```

```
$ ./first.sh
```

2 http://steve-parker.org/sh/eg/first.sh.txt

#!/ch/2

2. Philosophy

Shell script programming has a bit of a bad press amongst some Unix systems administrators. This is normally because of one of two things:

- The speed at which an interpreted program will run as compared to a C program, or even an interpreted Perl program.
- Since it is easy to write a simple batch-job type shell script, there are a lot of poor quality shell scripts around.

It is partly due to this that there is a certain machismo associated with creating *good* shell scripts. Scripts which can be used as CGI programs, for example, without losing out too much in speed to Perl (though both would lose to C, in many cases, were speed the only criterion).
There are a number of factors which can go into good, clean, quick, shell scripts.

- The most important criteria must be a clear, readable layout.
- Second is avoiding unnecessary commands.

A clear layout makes the difference between a shell script appearing as "black magic" and one which is easily maintained and understood.
You may be forgiven for thinking that with a simple script, this is not too significant a problem, but two things here are worth bearing in mind.

1. First, a simple script will, more often than anticipated, grow into a large, complex one.
2. Secondly, if nobody else can understand how it works, you will be lumbered with maintaining it yourself for the rest of your

life!

Something about shell scripts seems to make them particularly likely to be badly indented, and since the main control structures are if/then/else and loops, indentation is critical for understanding what a script does.

One of the major weaknesses in many shell scripts is lines such as:

```
cat /tmp/myfile | grep "mystring"
```

which would run much faster as:

```
grep "mystring" /tmp/myfile
```

Not much, you may consider; the OS has to load up the `/bin/grep` executable, which is a reasonably small 75600 bytes on my system, open a pipe in memory for the transfer, load and run the `/bin/cat` executable, which is an even smaller 9528 bytes on my system, attach it to the input of the pipe, and let it run.

Of course, this kind of thing is what the OS is there for, and it's normally pretty efficient at doing it. But if this command were in a loop being run many times over, the saving of not locating and loading the `cat` executable, setting up and releasing the pipe, can make some difference, especially in, say, a CGI environment where there are enough other factors to slow things down without the script itself being too much of a hurdle. Some Unices are more efficient than others at what they call "building up and tearing down processes" – i.e., loading them up, executing them, and clearing them away again. But however good your flavour of Unix is at doing this, it'd rather not have to do it at all.

As a result of this, you may hear mention of the Useless Use of Cat Award (UUoC), also known in some circles as **The Award For The Most Gratuitous Use Of The Word Cat In A Serious Shell Script** being bandied about on the `comp.unix.shell` newsgroup from time to time. This is purely a way of peers keeping each other in check, and making sure that things are done right.

Which leads me nicely on to something else: Don't *ever* feel too close to your own shell scripts; by their nature, the source cannot be closed. If you supply a customer with a shell script, s/he can inspect it quite easily. So you might as well accept that it will be inspected by anyone you pass it to; use this to your advantage with the GPL[3] - encourage people to give you feedback and bugfixes for free!

3 http://www.gnu.org/copyleft/gpl.html

#!/ch/3

3. A First Script

For our first shell script, we'll just write a script which says "Hello World". We will then try to get more out of a Hello World program than any other tutorial you've ever read :-)
Create a file (`first.sh`) as follows:

first.sh[4]

```
#!/bin/sh
# This is a comment!
echo Hello World    # This is a comment, too!
```

The first line tells Unix that the file is to be executed by `/bin/sh`. This is the standard location of the Bourne shell on just about every Unix system. If you're using GNU/Linux, `/bin/sh` is normally a symbolic link to bash (or, more recently, dash).

The second line begins with a special symbol: `#`. This marks the line as a comment, and it is ignored completely by the shell. The only exception is when the *very first* line of the file starts with `#!` - as ours does. This is a special directive which Unix treats specially. It means that even if you are using csh, ksh, or anything else as your interactive shell, that what follows should be interpreted by the Bourne shell. Similarly, a Perl script may start with the line `#!/usr/bin/perl` to tell your interactive shell that the program which follows should be executed by Perl. For Bourne shell programming, we shall stick to

4 http://steve-parker.org/sh/eg/first.sh.txt

#!/bin/sh.

The third line runs a command: `echo`, with two parameters, or arguments - the first is "`Hello`"; the second is "`World`".
Note that `echo` will automatically put a single space between its parameters.
The # symbol still marks a comment; the # and anything following it is ignored by the shell.

now run **`chmod 755 first.sh`** to make the text file executable, and run **`./first.sh`**.
Your screen should then look like this:

```
$ chmod 755 first.sh
$ ./first.sh
Hello World
$
```

You will probably have expected that! You could even just run:

```
$ echo Hello World
Hello World
$
```

Now let's make a few changes.
First, note that `echo` puts ONE space between its parameters. Put a few spaces between "Hello" and "World". What do you expect the output to be? What about putting a TAB character between them?
As always with shell programming, try it and see.
The output is exactly the same! We are calling the `echo` program with two arguments; it doesn't care any more than `cp` does about the gaps in between them. Now modify the code again:

```
#!/bin/sh
# This is a comment!
echo "Hello   World"    # This is a comment, too!
```

This time it works. You probably expected that, too, if you have experience of other programming languages. But the key to understanding what is going on with more complex command and shell

script, is to understand and be able to explain: *Why*?
echo has now been called with just one argument - the string "Hello   World". It prints this out exactly.
The point to understand here is that the shell parses the arguments before passing them on to the program being called. In this case, it strips the quotes but passes the string as one argument.
As a final example, type in the following script. Try to predict the outcome before you run it:

---

first2.sh[5]

```
#!/bin/sh
# This is a comment!
echo "Hello    World"     # This is a comment, too!
echo "Hello World"
echo "Hello * World"
echo Hello * World
echo Hello    World
echo "Hello" World
echo Hello "    " World
echo "Hello \"*\" World"
echo `hello` world
echo 'hello' world
```

Is everything as you expected? If not, don't worry! These are just some of the things we will be covering in this tutorial ... and yes, we will be using more powerful commands than echo!

5 http://steve-parker.org/sh/eg/first2.sh.txt

#!/ch/4

4. Variables - Part I

Just about every programming language in existence has the concept of *variables* - a symbolic name for a chunk of memory to which we can assign values, read and manipulate its contents. The Bourne shell is no exception, and this section introduces that idea. This is taken further in Variables - Part II which looks into variables which are set for us by the environment.

Let's look back at our first Hello World example. This could be done using variables (though it's such a simple example that it doesn't really warrant it!)

Note that there must be no spaces around the "=" sign: `VAR=value` works; `VAR = value` doesn't work. In the first case, the shell sees the "=" symbol and treats the command as a variable assignment. In the second case, the shell assumes that `VAR` must be the name of a command and tries to execute it.

If you think about it, this makes sense - how else could you tell it to run the command `VAR` with its first argument being "=" and its second argument being "`value`"?

Enter the following code into `var1.sh`:

---

var.sh[6]

```
#!/bin/sh
MY_MESSAGE="Hello World"
echo $MY_MESSAGE
```

---

6 http://steve-parker.org/sh/eg/var.sh.txt

This assigns the string "Hello World" to the variable `MY_MESSAGE` then `echo`es out the value of the variable.
Note that we need the quotes around the string Hello World. Whereas we could get away with `echo Hello World` because `echo` will take any number of parameters, a variable can only hold one value, so a string with spaces must be quoted to that the shell knows to treat it all as one. Otherwise, the shell will try to execute the command `World` after assigning `MY_MESSAGE=Hello`.

The shell does not care about types of variables; they may store strings, integers, real numbers - anything you like.
People used to Perl may be quite happy with this; if you've grown up with C, Pascal, or worse yet Ada, this may seem quite strange.
In truth, these are all stored as strings, but routines which expect a number can treat them as such.
If you assign a string to a variable then try to add 1 to it, you will not get away with it:

```
$ x="hello"
$ expr $x + 1
expr: non-numeric argument
$
```

This is because the external program `expr` only expects numbers. But there is no syntactic difference between:

```
MY_MESSAGE="Hello World"
MY_SHORT_MESSAGE=hi
MY_NUMBER=1
MY_PI=3.142
MY_OTHER_PI="3.142"
MY_MIXED=123abc
```

Note though that special characters must be properly escaped to avoid interpretation by the shell.
This is discussed further in Chapter 6, “Escape Characters”.

We can interactively set variable names using the `read` command; the following script asks you for your name then greets you personally:

var2.sh

```
#!/bin/sh
echo What is your name?
read MY_NAME
echo "Hello $MY_NAME - hope you're well."
```

Mario Bacinsky kindly pointed out to me that I had originally missed out the double-quotes in line 3, which meant that the single-quote in the word "you're" was unmatched, causing an error. It is this kind of thing which can drive a shell programmer crazy, so watch out for them!

This is using the shell-builtin command `read` which reads a line from standard input into the variable supplied.
Note that even if you give it your full name and don't use double quotes around the `echo` command, it still outputs correctly. How is this done? With the `MY_MESSAGE` variable earlier we had to put double quotes around it to set it.
What happens, is that the `read` command automatically places quotes around its input, so that spaces are treated correctly. (You will need to quote the output, of course - e.g. `echo "$MY_MESSAGE"`).

Scope of Variables

Variables in the Bourne shell do not have to be declared, as they do in languages like C. But if you try to read an undeclared variable, the result is the empty string. You get no warnings or errors. This can cause some subtle bugs - if you assign

```
  MY_OBFUSCATED_VARIABLE=Hello
```

and then

```
  echo $MY_OSFUCATED_VARIABLE
```

Then you will get nothing (as the second `OBFUSCATED` is mis-spelled).

There is a command called `export` which has a fundamental effect on the scope of variables. In order to really know what's going on with your variables, you will need to understand something about how this is used.

Create a small shell script, `myvar2.sh`:

myvar2.sh[7]

```
#!/bin/sh
echo "MYVAR is: $MYVAR"
MYVAR="hi there"
echo "MYVAR is: $MYVAR"
```

Now run the script:

```
$ ./myvar2.sh
MYVAR is:
MYVAR is: hi there
```

`MYVAR` hasn't been set to any value, so it's blank. Then we give it a value, and it has the expected result.
Now run:

```
$ MYVAR=hello
$ ./myvar2.sh
MYVAR is:
MYVAR is: hi there
```

It's still not been set! What's going on?!
When you call `myvar2.sh` from your interactive shell, a new shell is spawned to run the script. This is partly because of the `#!/bin/sh` line at the start of the script, which we discussed earlier.
We need to `export` the variable for it to be inherited by another program - including a shell script. Type:

```
$ export MYVAR
$ ./myvar2.sh
MYVAR is: hello
MYVAR is: hi there
```

7 http://steve-parker.org/sh/eg/myvar2.sh.txt

Now look at line 3 of the script: this is changing the value of `MYVAR`. But there is no way that this will be passed back to your interactive shell. Try reading the value of `MYVAR`:

```
$ echo $MYVAR
hello
$
```

Once the shell script exits, its environment is destroyed. But `MYVAR` keeps its value of `hello` within your interactive shell.
In order to receive environment changes back from the script, we must *source* the script - this effectively runs the script within our own interactive shell, instead of spawning another shell to run it.
We can source a script via the "." (dot) command:

```
$ MYVAR=hello
$ echo $MYVAR
hello
$ . ./myvar2.sh
MYVAR is: hello
MYVAR is: hi there
$ echo $MYVAR
hi there
```

The change has now made it out into our shell again! This is how your `.profile` or `.bash_profile` file works, for example.
Note that in this case, we don't need to `export MYVAR`.
Thanks to *sway* for pointing out that I'd originally said `echo MYVAR` above, not `echo $MYVAR` as it should be. Another example of an easy mistake to make with shell scripts. One other thing worth mentioning at this point about variables, is to consider the following shell script:

```
#!/bin/sh
echo "What is your name?"
read USER_NAME
echo "Hello $USER_NAME"
echo "I will create you a file called $USER_NAME_file"
touch $USER_NAME_file
```

Think about what result you would expect. For example, if you enter "steve" as your `USER_NAME`, should the script create `steve_file`? Actually, no. This will cause an error unless there is a variable called `USER_NAME_file`. The shell does not know where the variable ends and the rest starts. How can we define this?
The answer is, that we enclose the variable itself in *curly brackets*:

user.sh[8]

```
#!/bin/sh
echo "What is your name?"
read USER_NAME
echo "Hello $USER_NAME"
echo "I will create you a file called ${USER_NAME}_file"
touch "${USER_NAME}_file"
```

The shell now knows that we are referring to the variable `USER_NAME` and that we want it suffixed with "`_file`". This can be the downfall of many a new shell script programmer, as the source of the problem can be difficult to track down.

Also note the quotes around `"${USER_NAME}_file"` - if the user entered "Steve Parker" (note the space) then without the quotes, the arguments passed to `touch` would be `Steve` and `Parker_file` - that is, we'd effectively be saying `touch Steve Parker_file`, which is two files to be `touch`ed, not one. The quotes avoid this.

8 http://steve-parker.org/sh/eg/user.sh.txt

#!/more

Links To Other Resources

These are just a few of many other useful resources which can be found on the internet:

Steve Bourne's Introduction to the Unix Shell.
http://steve-parker.org/sh/bourne.shtml

A pretty definitive document on quoting and escape characters
http://www.mpi-inf.mpg.de/departments/rg1/teaching/unixffb-ss98/quoting-guide.html

What to watch out for when writing portable shell scripts.
http://archive09.linux.com/articles/34658

The Rosetta Stone for Unix :
http://bhami.com/rosetta.html

#!/more

The 600-Page Book

Steve Parker, the author of this tutorial, has also written a full book; Published by Wrox, the book "***Shell Scripting: Expert Recipes for Linux, Bash and more***" is available from all good retailers. ISBN 1118024486. RRP $49.99. Also available on Kindle:

http://steve-parker.org/book/ has the details; the book is available to buy online and in stores.

Steve Parker has also written "***How to Build a LAMP Server***", which details the installation of the LAMP stack, and building a working application on top of it, including MySQL configuration, phpMyAdmin use, and PHP development. This is currently only available on Amazon as an e-Book; search for "B00DAI6ATO", or for "How to Build a LAMP Server".

See http://steve-parker.org/publications/ for the latest information.