Classic Shell Scripting (Book Review)

By John Weatherwax

BKCSS.RVW 20061021

```
"Classic Shell Scripting", Arnold Robbins & Nelson Beebe, 2005,
0-596005-95-4
%A   Arnold Robbins
%A   Nelson H. F. Beebe
%D   February 1, 2005
%G   0-596005-95-4
%I   O'Reilly Media
%O   http://www.amazon.com/Classic-Shell-Scripting-Arnold-Robbins[...]
%P   534 pages
%T   Classic Shell Scripting
```

When I first started reading "Classic Shell Scripting" by Arnold Robbins and Nelson H. F. Beebe, the quality of the content inspired me to write a review of this book. As opposed to most books on the subject that only explain and give examples of syntax, this book aims to develop in the reader a deeper understanding and true mastery of the POSIX shell.

The UNIX toolbox philosophy has been (to use a description from Robert Floyd's acceptance of the ACM Turing Award) a staple "Programming Paradigm" for UNIX programmers for several decades. In addition, to developing a better understanding of existing UNIX tools, this book will help programmers understand the ideas behind "pipeline programming": producing programs that take data sources, process the data in a sequence of serially connected steps and output the end result.

For example, the "wc" command provides standard counts of characters, words, and line numbers in a given file. To count all of the non-comment lines in all shell scripts (assumed to end with .sh) in ~/bin:

```
cat `find ~/bin/ -name "*.sh"` | sed -e '/^ *#/d' | wc -l
```

This simple pipeline creates a data source containing all lines from all shell scripts ending with .sh in your ~/bin directory. The sed command then deletes any lines beginning with comments and the wc command counts the remaining lines. Commands like this make it easy to report how many lines of code an application has. This is a simple example of what can be done with UNIX pipelines. This book helps the reader develop the skills to write such programs.

The book begins with an inviting preface and draws the reader in, right away. The authors mention that the point of learning shell scripting is to obtain proficiency in using the UNIX toolbox philosophy. This entails using a combination of smaller specialized tools to accomplish a larger task. As such, the book focuses on several main themes:

- What the Unix tools are, specifically the reasons why the various commands were created and the special jobs they were intended to do.
- How to combine these Unix tools using pipes and file redirections.
- Popular extensions to standard tools, specifically the use of GNU- or BSD-derived Unix systems and some indispensable nonstandard shell tools.

Each chapter contains background and introductions into the Unix toolbox philosophy. Collectively, they emphasize the need for and eventual standardization of the Unix utilities through the POSIX standard, the main points of which are:

- Each command should do a single task (only) and do it as well as possible.
- Process input and output lines of text, not any specific binary format.
- A thorough understanding of regular expressions will ease many text processing tasks.
- Default input and output comes from the keyboard and goes to the console respectively.
- A program should do its task as quietly as possible and operate with no unnecessary output.
- Borrow from the work of others: Learn as much as possible about existing commands and use them whenever possible. If no command exists to do the task at hand ... invent one. This is the focus of this book. Use the commands already in existence to construct new ones.

[Much of the above recapitulates the "Main Tenets" from Mike Gancarz' "The UNIX Philosophy" (Digital Press, Newton, MA 1995, 151 pp., $19.95, ISBN 1-55558-123-4) - another excellent read. -- Ben]

I found the book to be a plethora of interesting ideas and command descriptions. Rather than describe each chapter in detail, I have chosen to present a sequence of "factoids" containing the "Classic Shell Scripting" content I found most interesting. I should mention that these are just a sample of the types of things you can learn by reading this book.

Portability of "echo"

The built-in command echo is not as universal as you might first think. The BSD version allowed the switch "-n" to disable the printing of a newline after the string:

```
echo -n "Enter Choice: "
```

The System V version of echo used another approach for the same purpose: They choose to implement a version that recognized a special escape sequence "\c", so the above would become:

```
echo "Enter Choice: \c"
```

The current POSIX standard on echo does not include the "-n" option. On my Linux system there are two echo commands: one in the shell and other located in /bin. The System V behavior can be implemented with a "-e" switch

```
echo -e "Enter Choice: \c"
```

The purpose of this discussion is that echo in shell scripts may not be as portable as one imagines. For very simple string printing, this is not usually a problem. For more complicated situations, one should use the POSIX standardized command "printf". To do the above with printf, one would use (as the newline is provided by default):

```
printf "Enter Choice: "
```

If a newline was desired, it could be inserted with "\n".

Debugging Shell Scripts:

Debugging shell scripts can be a simple as inserting a "-x" in the shebang head of a shell script. For instance, replace

```
#!/bin/sh
```

with

```
#!/bin/sh -x
```

The "-x" flag results in the shell echoing each and every command before the command gets executed. Each sub-shell created also increments a prompt, so you can tell at what stack level each command executed. For instance, if your script is

```
#!/bin/sh

cat $1
echo `wc $1`
```

Given an input set of files (file{1,2,3}) like the following (the $ being the shell prompt)

```
$cat file1
file2
$cat file2
file3
$cat file3
Finally some data!
```

the script (degEg.sh):

```
#!/bin/sh -x

cat file1
cat $(cat file1)
cat $(cat $(cat file1))
```

produces for output

```
+ cat file1
file2
++ cat file1
+ cat file2
file3
+++ cat file1
++ cat file2
+ cat file3
Finally some data!
```

Each level of "+"'s denotes the stack level in the script. For instance, the first command "cat file1" is at stack level 1 and produces the result "file2". The next command is

```
cat $(cat file1)
```

which must execute cat file1 first, before it can execute the "cat" command on the existing result. This "inner" call is performed at stack level 2, represented in the above as

```
++ cat file1
```

this result is file2, and is subject to the next cat command given in debugging output like

```
+ cat file2
```

with the result of "file3". The rest of the example is similar.

Internationalization of your commands

There has been a recent push in the POSIX community aimed at internationalization. As such, you can make your computer speak Italian and display help for the ls command, with

```
LC_ALL=it_IT ls --help
```

These are just a small sample of some of the interesting things this book has to offer. If you are a shell programmer who wants to take his/her skills "to the next level", you should consider reading this book.