The Taming of the Shell

Tom Ryder
tom@sanctum.geek.nz
https://sanctum.geek.nz/

What is the shell?

- The **command-line shell** is a text-based, line-driven interpreter for building, executing, and managing commands on the system.
- On GNU/Linux, it's almost always GNU Bash (bash)
  - Sometimes Z-shell (`zsh`) for advanced users
  - The system shell `/bin/sh` on some Linuces, including Debian, is a stripped-down POSIX shell called `dash`
- Don't confuse it with your **terminal emulator**.
  - `gnome-terminal`, `xfce-terminal`, the TTY...

Not really a tutorial

- I can't teach you good shell script in half an hour.
- But I *can*...
  - Make you less wary of and more excited by it
  - Point you to the resources I find useful to write decent shell
  - Show you where it's useful
  - Warn you where it isn't
  - Demonstrate what I do with it

Lineage

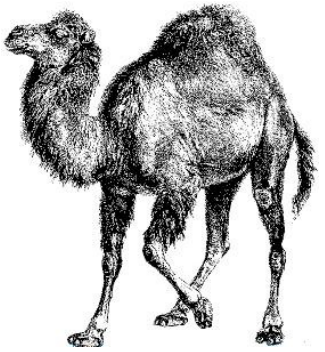
- The Multics shell (pre-1969)
  - Part of an abandoned OS project at Bell Labs
- The Thompson shell (Unix v1-6, 1971-1975)
  - **Ken Thompson**, that is
  - Pipes, some control structures, some wildcards
- The PWB/Mashey shell (1975-1979)
  - Variables
  - **Shell scripts**
- **The Bourne Shell**
  - This is where things got *really* interesting
  - **GNU Bash**, **Korn shell**, and **Zsh** are all essentially souped-up Bourne shells
  - Syntax from this family of shells standardised into POSIX

Shell families

- The Bourne family of shells is by far the biggest, and is standardised in POSIX.
  - If you know a little Bash, this is what you know.
- You could use a shell from the mostly-incompatible **C shell** syntax family instead...
  - ...but that’s an uphill battle that you might regret, unless you actually have to support C shell.
  - It used to be very, very broken, and required really nasty workarounds, though it’s not as bad as it used to be.
  - Almost every shell example or discussion you will find online will default to Bourne family shell support, mostly GNU Bash.

Shell script is a dying art

- The shell itself is still used plenty, but there's a lot of **cargo-culting** going on.
- People just run the scripts provided by their normal software packages, or copy-paste commands from the internet, adapting their paths as appropriate.
- Those scripts are often cribbed from other packages.
- Shell script is seen as *dangerous*, *broken*, *unusable*, *arcane*.
- Worse than **Perl**...
- ...but we'll get to that.



The Unix Philosophy

1) Write programs that do one thing and do it well.

2) Write programs to work together.

3) Write programs to handle text streams, because that is a universal interface.

Following these rules makes for good shell script, too.

What went wrong?

- **Rob Pike** came later to the Unix team, and we get some hints from him:
  - “Those days are dead and gone, and the eulogy was delivered by Perl.”
  - “Let the whiners whine: you're right, and they don't know what they're missing. Unfortunately, I do, and I miss it terribly.”
  - “The Unix room still exists, and it may be the greatest cultural reason for the success of Unix as a technology.”

    https://en.wikiquote.org/wiki/Rob_Pike

Virtues of shell script

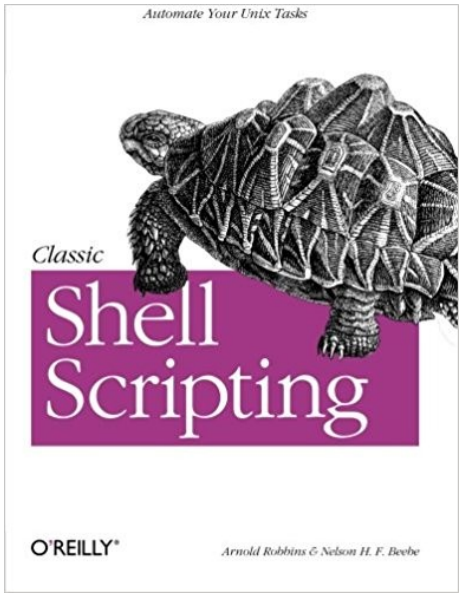
- At a **basic level**, you start by simply wrapping commands you often run in a single well-named script, and whack it somewhere in your $PATH for private use.
  - Rsync invocations
  - Git or Subversion arcana
  - Key and certificate management
  - Deploying websites
  - ...and other painful stuff not worth memorising
- This is a fantastic productivity booster all by itself.
- Even if you never get past this step, you’re still ahead of the game if you do it consistently.

Virtues of shell script

- That means you develop a lot of **monolithic scripts** that do one very precise, system or application-specific thing.
- As you get more **advanced**, you find repeated and more specific operations you often do on the command line.
- You start writing your own **private general tool set**, and it’s different for everyone:
  - Transforming text: changing case, removing non-printable characters...
  - Selecting columns...
  - Skipping lines...
  - Wrapping processes...
  - Presenting your to-do list on login...
  - Automating your job so well that nobody actually notices...

Shell script life cycle

- Someone else on the system finds your script **useful**
- You **promote** it into `/usr/local/bin`
- You **rewrite** it
- You write a **manual page** because people keep asking you how to use it
- You get drunk one night and put it on **GitHub** with a free license
- Some nerd packages it into **Debian** without asking you
- People report **bugs**
- You rewrite it **again**
- You buy a book with a **tortoise** on the cover
- You start growing a **beard**



What is it good for?

- **Running other programs**
  - In sequence, in parallel, conditionally...
- Simple process management
- Saving, redirecting, piping, or silencing program output and errors
- Looping over arguments to repeat some task
  - Especially filenames
- Simple pattern matching
  - Especially filenames
- Simple variable logic
  - Especially filenames

Think of shell script as a **glue language**; it's just smart enough to put together scripts written in other languages in clever ways, especially string-chomping languages like `sed`, `awk`, and `perl`.

What does it suck at?

- A first programming language
  - If you're totally new, learn Python first
- Maths
  - POSIX shell and Bash can only do integers
  - Use `awk` or `bc` instead
- Data structures
  - POSIX shell has string variables and an argument list, and that's basically it
  - Use C, Perl, or Python instead
- Speed of execution
  - Use literally anything else
  - Put the slow bits in a fast language
- Long programs
  - 100 lines is probably the sanity limit

Which shell for **interactive use?**

- It's entirely up to you. `$SHELL` choice is personal, like `$EDITOR` choice.
- Use whatever makes the most sense to you.
- You can still run `#!shebanged` shell scripts in any language.

That said...

- Choosing a common shell means you won't have to install it (or ask the sysadmin to install it) on new systems.
- It does help to use a Bourne-family shell, if that's what you script in too.
  - There are lots of Zsh zealots around...
  - Some people even use Fish, the Friendly Interactive Shell...
  - Me? I just use Bash.

Which shell for **scripting**?

Well, not that I’m biased, but...

- **POSIX shell**

  POSIX shell script,
  Or no shell script at all!
  Wimps and poseurs,
  Leave the hall!

Which shell for **scripting**?

- Seriously though, it **depends on the platform**.
- If you run into walls with the POSIX feature set:
  - ...and the script doesn’t need to leave a GNU/Linux machine, **just use Bash**.
  - ...if there will always be a Korn shell, **just use Korn shell**.
- After all, you can always port it later.
- Even hardcore BSD admins seem to install Bash these days.

Obscure syntax

What does this do?

```
:(){ :|: & };:
```

Obscure syntax

- Here's that **cargo-culting** we talked about.
- A friend's colleague wrote this:

```
q=$1
mactest=`echo "$q" | grep -o ":" | wc -l`
if [[ "$mactest" -ne "0" ]] ; then
...
fi
```

- Which is much easier like this:

```
case $1 in *:*) ... ;; esac
```

How to do it right?

- **Use ShellCheck**: https://www.shellcheck.net/
- **Keep it short and simple.**
- Don't trust Stack Overflow.
- Be wary of Google.
- Don't imitate `./configure` scripts or OS shell scripts.

Reference material

- Greg's Wiki:
  - Guide: http://mywiki.wooledge.org/BashGuide
  - FAQ: http://mywiki.wooledge.org/BashFAQ
  - Pitfalls: http://mywiki.wooledge.org/BashPitfalls
- The Bash Hackers Wiki:
  - http://wiki.bash-hackers.org/
- The POSIX standard:
  - http://pubs.opengroup.org/onlinepubs/9699919799/

Some common problems

- You might see someone chopping off the leading path of a filename like this:

  ```
  b=`echo $f | sed "s/.*\/\(.*\)/\1/"`
  ```

  Yikes!

- A little better:

  ```
  b=$(printf %s "$f" | sed 's_.*/__')
  ```

- A *lot* better:

  ```
  b=$(basename -- "$f")
  ```

- Optimal (no forks, no whitespace edge case):

  ```
  b=${f##*/}
  ```

Some common problems

- You have a line like this in your script:
  `rm -fr $file_to_remove`
  - One day, someone sets `file_to_remove=*`...
  - Oh no, where are all your files?!
- You need to **double-quote** all your variables:
  `rm -fr "$file_to_remove"`
- And you need to ensure they won't be parsed as **options**:
  `rm -fr -- "$file_to_remove"`

Demonstration

- We'll look at some simple POSIX shell scripts written by the presenter (Tom).
  - Shortcuts
  - String filters
  - Execution wrappers
  - Documentation
  - Installation (`Makefile`)
- Please ask questions if anything looks interesting.
- All of this code is freely available with a public domain license: https://sanctum.geek.nz/cgit/dotfiles.git/tree/bin

Questions?

Email: tom@sanctum.geek.nz

Web: https://sanctum.geek.nz/

The presenter loves this topic in particular, and would be happy to present again on anything that looked interesting to the audience.

Command line hacks, `awk`, `sed`, `perl`, etc...