Don’t Write Your Bash Scripts, Generate Them!

Matěj Týč, 01/27/2019
Outline

Learn how to write scripts that meet and exceed expectations with minimal effort.

- The theory: Good-behaved scripts.
- The observation: Code generators.
- The experiment: Live demo!
CLI awesomness I

- Optional args: -o [value]
- Positional args: first second third

- $ ls -l -A -w 10 ./-x foo bar
- $ ls -lA -w10 ./-x foo bar
- $ ls -lAw10 -- -x foo bar
Tools of the trade

- `getopt`: external utility, GNU version exists.
- `getopts`: Bash builtin, cross-platform.
CLI awesomeness II

- Optional args:
  - `--option [=value]`  `--option [value]`
  - `-o [value]`
- Positional args: `first [opt args ...]`  `second`  `third`
- `$ ls -l foo -A --width=10 bar ./-x`
- `$ ls -lw10 --almost-all foo bar ./-x`
- `$ ls -lAw10 -- foo bar -x`
Non-bash

There are plenty of arg parsing libraries:

<table>
<thead>
<tr>
<th>Language</th>
<th>Utility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>Python argparse</td>
</tr>
<tr>
<td>Perl</td>
<td>Getopt::Long</td>
</tr>
<tr>
<td>C</td>
<td>Getopt, popt, C argparse</td>
</tr>
<tr>
<td>C++</td>
<td>Boost::ProgramOptions, QCommandLineParser</td>
</tr>
<tr>
<td>Lua</td>
<td>Lua Argparse</td>
</tr>
<tr>
<td>Go</td>
<td>Flag</td>
</tr>
</tbody>
</table>
How do I parse command line arguments in Bash?

Say, I have a script that gets called with this line:

```
./myscript -vfd ./foo/bar/someFile -o /fizz/someOtherFile
```

or this one:

```
./myscript -v -f -d -o /fizz/someOtherFile ./foo/bar/someFile
```

What's the accepted way of parsing this such that in each case (or some combination of the two)

- `$v`, `$f`, and `$d` will all be set to `true` and
- `$outFile` will be equal to `/fizz/someOtherFile`?
How do I parse command line arguments

Say, I have a script that gets called with this line:

```
./myscript -vfd ./foo/bar/someFile -o /fizz/someOtherFilename
```

or this one:

```
./myscript -v -f -d -o /fizz/someOtherFile ./foo/bar/someOtherFile
```

What's the accepted way of parsing this such that in each case ($v$, $f$, and $d$ will all be set to true and $o$ will be equal to 

bash command-line scripting arguments
Preferred Method: Using straight bash without getopt[s]

I originally answered the question as the OP asked. This Q/A is getting a lot of attention, so I should also offer the non-magic way to do this. I'm going to expand upon guneysus's answer to fix the nasty sed and include Tobias Kienzler's suggestion.

Two of the most common ways to pass key value pair arguments are:

**Straight Bash Space Separated**

Usage  
```
./myscript.sh -e conf -s /etc -l /usr/lib /etc/hosts
```

```bash
#!/bin/bash
# Use -gt 1 to consume two arguments per pass in the loop (e.g. each
# argument has a corresponding value to go with it).
# Use -gt 0 to consume one or more arguments per pass in the loop (e.g.
```
Preferred Method: Using straight bash without getopt[s]

I originally answered the question as the OP asked. This Q/A is getting a lot of attention, so I should also offer the non-magic way to do this. I'm going to expand upon cuneusus's answer to fix the nasty

Method #1: Using bash without getopt[s]

Two common ways to pass key-value-pair arguments are:

Bash Space-Separated (e.g., --option argument) (without getopt[s])

Usage ./.myscript.sh -e conf -s /etc -l /usr/lib /etc/hosts

#!/bin/bash

POSITIONAL=()
while [[ $# -gt 0 ]]
do
How can I handle command-line arguments (options) to my script easily?

Well, that depends a great deal on what you want to do with them. There are several approaches, each with its strengths and weaknesses.

Contents
1. How can I handle command-line arguments (options) to my script easily?
   1. Manual loop
   2. getopts

**Manual loop**

Manually parsing options without the use of a specialized function is the most flexible approach, and is sufficient for most simple scripts.

This example will handle a combination of short (POSIX) and long "GNU style" options with option arguments. Notice how both --file FILE and scripts may also use functions and local variables, which can greatly improve your code. This example however illustrates a strictly POSIX co

```bash
#!/bin/sh
# POSIX

# Reset all variables that might be set
file=
verbose=0  # Variables to be evaluated as shell arithmetic should be initialized to a default or validated

while ;; do
  case $1 in
    --help) show help
      # Call a "show_help" function to display a synopsis, then exit.
    esac
```

...
Why code generators

• Bash modules ...
  – are pain to distribute for users.
  – create a huge external dependency.
  – usually have bad documentation.
• Builtins are horrible to use.
• Manpage, help, completion code duplication.
Why argbash

• Code generators:
  – Need to be installed.
  – Have to be able to regenerate modified script.
  – Should better generate nice code.
• You still have to ship the generated code.
Why use argbash

- Generated code is a template.
- Available as:
  - Local install: Github, Fedora, AUR
  - Docker image: Search the docker hub
  - Online service https://argbash.io/generate
- Feature-rich and documented.
Why use argbash

- Lots of outputs: Bash script, POSIX script, bash completion, manpage, docopt.
- Generates simple code for simple scripts, complex code for complex ones.
- Commented mode for greater usability.
Demo time!
Thank you attention!

Don’t forget to https://argbash.io

Any questions?