I've talked before about how I am a lazy shell script programmer. It might be because I'm simply not a full-time professional software developer, and I don’t even administer my own servers anymore—I outsource the job to Wisconsin.

Regardless of how much I program nowadays though, I still find myself needing simple little applications—tiny programs that do one simple task well.

And, then there are the throwaway scripts that stick around, ultimately becoming a mainstay of one’s toolkit, spreading out to cover multiple functions and mysteriously growing to 100 lines or more.

I have one of those in my toolkit, a script that originally was intended simply to figure out the dimensions of a graphic file and produce the proper height and width attributes for an HTML image tag.

Now the script scale.sh has grown to 133 lines and does a variety of different, albeit related tasks. No surprise, it’s also grown to have a variety of command-line arguments, as shown here:

```bash
$ ./scale.sh
```

Usage: scale [args] factor [file or files]
- a use URL values for Apparenting.com site
- b add 1px solid black border around image
- i use URL values for Intuitive.com/blog site
- k KW add keywords KW to the ALT tags
- r use ‘align=right’ instead of <center>
- s produces succinct dimensional tags only

A factor 0.9 for 90% scaling, 0.75 for 75%, or max width in pixels.
A factor of ‘1’ produces 100%.

Crack open the code, and you’ll see my dirty little scripting secret—a very sloppy approach to parsing command-line options:

```bash
if [ "$1" = "-a" ] ; then
    baseurl="www.apparenting.com/Images/"; shift
fi
```

I did warn you that I was a lazy programmer, right? This is a pretty classic way to parse and process command-line arguments, actually. Check the value of $1, and if it’s a known flag, change a default variable or two, then use the shift command to move $2 → $1, $3 → $2 and so on, effectively deleting the processed flag from the command-line args.

The problem is, when you have more than one or two flags, this really doesn’t work. I step through the command flags alphabetically in my script—for example, invoking the script as `scale -r -a` will fail. It'll process the -r flag but never see the -a and generate an error condition.
Fortunately, there's a very nice Linux command called getopt that lets you parse through your command flags in a far more sophisticated manner.

**getopt In Shell Scripts**

The getopt command first requires that you let it rearrange how your command flags are organized, then you use the set command to update all the positional variables. After that, you can step through the positional variables with a case statement.

The first step is:

```bash
args=`getopt FLAGS $*`
set -- $args
```

where FLAGS should be the individual letters of known and accepted command flags. If a flag has an argument that goes with it (like -s 30), append a colon to it.

For my script, it looks like this:

```bash
args=`getopt abik:rs $*`
set -- $args
```

To see what happens, I've added a bonus echo statement. Here's the result:

```bash
$ scale -abs -k fdsf 100 *png
args = -a -b -s -k fdsf -- 100 blooeh.png
```

As you can see, getopt separates out each and every command flag and adds a -- flag that indicates when the command flags end—simple, really!

Now that the args have been restructured, parsing is relatively easy, though it looks pretty complicated (warning, I've stripped out a few clauses for simplicity):

```bash
for i; do
  case "$i" in
    -a  ) baseurl="www.apparenting.com/Images/"
      shift ;;
    -k  ) keywords=" ($2)"
      shift ; shift ;;
    -s  ) verbose=0
      shift ;;
    --  ) shift; break ;;
  esac
done
```

Let's read this backward. At the -- option, the loop will exit due to the break. Until that's hit, the for loop will just keep iterating, stepping through all the flags specified. This is how the order of the flags becomes irrelevant.

Each time a flag is matched, the desired action is taken, variables are set and so on, then the shift command shows up again to move all the command flags down one (for example, $2 to $1, $3 to $2 and so on).

Shell script case statement matching lines are all in the form of:

```bash
regex ) actions ;;
```

The double semicolon is an oddity, but that's how you indicate the end of an individual case match, hence the notation shown above.

Grabbing the argument for the -k flag is easy too, because getopt has made sure that it's a separate argument, and since we're using shift as we go along to move things around, $2 will always be the argument itself.

Finally, also notice that as a stylistic approach, I have the double semicolon with a leading space. That's just so when I eyeball the script, I quickly can recognize if there are any cases that are missing the double semicolon.

The only piece missing is some error handling, because right now, if a bad flag is encountered, here's what happens:

```bash
$ scale -ax 100 *png
getopt: illegal option -- x
```

Nice, but the script doesn't catch the error condition or stop running—not so good.
To fix it, immediately after the call to getopt, simply test the return code:

```bash
if [ $? != 0 ] ; then ...
```

In the conditional, you probably would put a usage statement and an exit command. For my script, I actually also test to ensure that there are a minimum of two arguments on the command line as well, because the script is never valid without them:

```bash
if [ $? != 0 -o $# -lt 2 ] ; then
  echo "Usage: scale {args} factor [file or files]"
  exit 0
fi
```

At this point in our shell script writing journey, I certainly hope you can read that rather cryptic conditional statement and understand what it does.

Ultimately, it's a bit of work to parse command-line flags the right way, but it makes for a far more flexible and robust shell script.

Dave Taylor has been involved with UNIX since he first logged in to the on-line network in 1980. That means that, yes, he's coming up to the 30-year mark now. You can find him just about everywhere on-line, but start here: www.DaveTaylorOnline.com.