Small getopts tutorial

Description

When you want to parse command line arguments in a professional way, `getopts` is the tool of choice. Unlike its older brother `getopt` (note the missing *s*!), it's a shell builtin command. The advantages are:

- No need to pass the positional parameters through to an external program.
- Being a builtin, `getopts` can set shell variables to use for parsing (impossible for an *external* process!)
- There's no need to argue with several `getopt` implementations which had buggy concepts in the past (whitespace, …)
- `getopts` is defined in POSIX®.

Some other methods to parse positional parameters (without `getopt(s)`) are described in: How to handle positional parameters.

Note that `getopts` is not able to parse GNU-style long options (`--myoption`) or XF86-style long options (`-myoption`)!

Terminology

It's useful to know what we're talking about here, so let's see… Consider the following command line:

```
mybackup -x -f /etc/mybackup.conf -r ./foo.txt ./bar.txt
```

These are all positional parameters, but they can be divided into several logical groups:

- `-x` is an **option** (aka **flag** or **switch**). It consists of a dash (`-`) followed by **one** character.
- `-f` is also an option, but this option has an associated **option argument** (an argument to the option `-f`): `/etc/mybackup.conf`. The option argument is usually the argument following the option itself, but that isn't mandatory. Joining the option and option argument into a single argument `-f/etc/mybackup.conf` is valid.
- `-r` depends on the configuration. In this example, `-r` doesn't take arguments so it's a standalone option like `-x`.
- `./foo.txt` and `./bar.txt` are remaining arguments without any associated options. These are often used as **mass-arguments**. For example, the filenames specified for `cp(1)`, or arguments that don't need an option to be recognized because of the intended behavior of the program. POSIX® calls them **operands**.

To give you an idea about why `getopts` is useful, The above command line is equivalent to:

```
mybackup -xrf /etc/mybackup.conf ./foo.txt ./bar.txt
```

which is complex to parse without the help of `getopts`.

The option flags can be **upper- and lowercase** characters, or **digits**. It may recognize other characters, but that's not recommended (usability and maybe problems with special characters).

How it works

In general you need to call `getopts` several times. Each time it will use the next positional parameter and a possible argument, if parsable, and provide it to you. `getopts` will not change the set of positional parameters. If you want to shift them, it must be done manually:

```
shift $((OPTIND-1))

# now do something with $@
```

Since `getopts` sets an exit status of *FALSE* when there's nothing left to parse, it's easy to use in a while-loop:

```
while getopts ...; do

  ...

done
```

`getopts` will parse options and their possible arguments. It will stop parsing on the first non-option argument (a string that doesn't begin with a hyphen (`-`) that isn't an argument for any option in front of it). It will also stop parsing when it sees the `--` (double-hyphen), which means end of options.

Used variables

variable	description
OPTIND	Holds the index to the next argument to be processed. This is how `getopts` "remembers" its own status between invocations. Also useful to shift the positional parameters after processing with `getopts`. `OPTIND` is initially set to 1, and **needs to be re-set to 1 if you want to parse anything again with getopts**
OPTARG	This variable is set to any argument for an option found by `getopts`. It also contains the option flag of an unknown option.
OPTERR	(Values 0 or 1) Indicates if Bash should display error messages generated by the `getopts` builtin. The value is initialized to **1** on every shell startup - so be sure to always set it to **0** if you don't want to see annoying messages! **`OPTERR` is not specified by POSIX for the `getopts` builtin utility — only for the C `getopt()`**

variable	description
	function in `unistd.h` (`opterr`). `OPTERR` is bash-specific and not supported by shells such as ksh93, mksh, zsh, or dash.

`getopts` also uses these variables for error reporting (they're set to value-combinations which arent possible in normal operation).

Specify what you want

The base-syntax for `getopts` is:

```
getopts OPTSTRING VARNAME [ARGS...]
```

where:

`OPTSTRING`	tells `getopts` which options to expect and where to expect arguments (see below)
`VARNAME`	tells `getopts` which shell-variable to use for option reporting
`ARGS`	tells `getopts` to parse these optional words instead of the positional parameters

The option-string

The option-string tells `getopts` which options to expect and which of them must have an argument. The syntax is very simple — every option character is simply named as is, this example-string would tell `getopts` to look for `-f`, `-A` and `-x`:

```
getopts fAx VARNAME
```

When you want `getopts` to expect an argument for an option, just place a `:` (colon) after the proper option flag. If you want `-A` to expect an argument (i.e. to become `-A SOMETHING`) just do:

```
getopts fA:x VARNAME
```

If the **very first character** of the option-string is a `:` (colon), which would normally be nonsense because there's no option letter preceding it, `getopts` switches to "**silent error reporting mode**". In productive scripts, this is usually what you want because it allows you to handle errors yourself without being disturbed by annoying messages.

Custom arguments to parse

The `getopts` utility parses the positional parameters of the current shell or function by default (which means it parses `"$@"`).

You can give your own set of arguments to the utility to parse. Whenever additional arguments are given after the `VARNAME` parameter, `getopts` doesn't try to parse the positional parameters, but these given words.

This way, you are able to parse any option set you like, here for example from an array:

```
while getopts :f:h opt "${MY_OWN_SET[@]}"; do
  ...
done
```

A call to `getopts` **without** these additional arguments is **equivalent** to explicitly calling it with `"$@"`:

```
getopts ... "$@"
```

Error Reporting

Regarding error-reporting, there are two modes `getopts` can run in:

- verbose mode
- silent mode

For productive scripts I recommend to use the silent mode, since everything looks more professional, when you don't see annoying standard messages. Also it's easier to handle, since the failure cases are indicated in an easier way.

Verbose Mode

invalid option	`VARNAME` is set to `?` (question-mark) and `OPTARG` is unset
required argument not found	`VARNAME` is set to `?` (question-mark), `OPTARG` is unset and an *error message is printed*

Silent Mode

invalid option	`VARNAME` is set to `?` (question-mark) and `OPTARG` is set to the (invalid) option character
required argument not found	`VARNAME` is set to `:` (colon) and `OPTARG` contains the option-character in question

Using it

A first example

Enough said - action!

Let's play with a very simple case: only one option (`-a`) expected, without any arguments. Also we disable the *verbose error handling* by preceding the whole option string with a colon (`:`):

```
#!/bin/bash
```

```
while getopts ":a" opt; do
  case $opt in
    a)
      echo "-a was triggered!" >&2
      ;;
    \?)
      echo "Invalid option: -$OPTARG" >&2
      ;;
  esac
done
```

I put that into a file named `go_test.sh`, which is the name you'll see below in the examples.

Let's do some tests:

Calling it without any arguments

```
$ ./go_test.sh
$
```

Nothing happened? Right. `getopts` didn't see any valid or invalid options (letters preceded by a dash), so it wasn't triggered.

Calling it with non-option arguments

```
$ ./go_test.sh /etc/passwd
$
```

Again — nothing happened. The **very same** case: `getopts` didn't see any valid or invalid options (letters preceded by a dash), so it wasn't triggered.

The arguments given to your script are of course accessible as `$1` - `${N}`.

Calling it with option-arguments

Now let's trigger `getopts`: Provide options.

First, an **invalid** one:

```
$ ./go_test.sh -b
Invalid option: -b
$
```

As expected, `getopts` didn't accept this option and acted like told above: It placed `?` into `$opt` and the invalid option character (`b`) into `$OPTARG`. With our `case` statement, we were able to detect this.

Now, a **valid** one (`-a`):

```
$ ./go_test.sh -a

-a was triggered!

$
```

You see, the detection works perfectly. The `a` was put into the variable `$opt` for our case statement.

Of course it's possible to **mix valid and invalid** options when calling:

```
$ ./go_test.sh -a -x -b -c

-a was triggered!

Invalid option: -x

Invalid option: -b

Invalid option: -c

$
```

Finally, it's of course possible, to give our option **multiple times**:

```
$ ./go_test.sh -a -a -a -a

-a was triggered!

-a was triggered!

-a was triggered!

-a was triggered!

$
```

The last examples lead us to some points you may consider:

- **invalid options don't stop the processing**: If you want to stop the script, you have to do it yourself (`exit` in the right place)
- **multiple identical options are possible**: If you want to disallow these, you have to check manually (e.g. by setting a variable or so)

An option with argument

Let's extend our example from above. Just a little bit:

- `-a` now takes an argument

- on an error, the parsing exits with `exit 1`

```
#!/bin/bash


while getopts ":a:" opt; do
  case $opt in
    a)
      echo "-a was triggered, Parameter: $OPTARG" >&2
      ;;
    \?)
      echo "Invalid option: -$OPTARG" >&2
      exit 1
      ;;
    :)
      echo "Option -$OPTARG requires an argument." >&2
      exit 1
      ;;
  esac
done
```

Let's do the very same tests we did in the last example:

Calling it without any arguments

```
$ ./go_test.sh
$
```

As above, nothing happened. It wasn't triggered.

Calling it with non-option arguments

```
$ ./go_test.sh /etc/passwd
$
```

The **very same** case: It wasn't triggered.

Calling it with option-arguments
Invalid option:

```
$ ./go_test.sh -b
Invalid option: -b
```

```
$
```

As expected, as above, `getopts` didn't accept this option and acted like programmed.

Valid option, but without the mandatory **argument**:

```
$ ./go_test.sh -a
Option -a requires an argument.
$
```

The option was okay, but there is an argument missing.

Let's provide **the argument**:

```
$ ./go_test.sh -a /etc/passwd
-a was triggered, Parameter: /etc/passwd
$
```