Small getopts tutorial

Description

Note that `getopts` is neither able to parse GNU-style long options (`--myoption`) nor XF86-style long options (`-myoption`). So, when you want to parse command line arguments in a professional 😉 way, `getopts` may or may not work for you. 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 - using neither **getopt** nor **getopts** - are described in: How to handle positional parameters.

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.

variable	description
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()` 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
$
```

See also

- Internal: Handling positional parameters
- Internal: The case statement
- Internal: The while-loop
- POSIX getopts(1) (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/getopts.html#tag_20_54) and getopt(3) (http://pubs.opengroup.org/onlinepubs/9699919799/functions/getopt.html)
- parse CLI ARGV (https://stackoverflow.com/questions/192249/how-do-i-parse-command-line-arguments-in-bash)
- handle command-line arguments (options) to a script (http://mywiki.wooledge.org/BashFAQ/035)

Discussion

Steve Lessard, 2010/07/14 15:04 ()

Thanks for the very nice tutorial. I've learned a lot from it. However one thing appears to be missing in this tutorial, how to use the optional 'ARGS' argument. You gave a very brief description of what 'ARGS' does but didn't give an explanation of how it works. What does it mean to parse "words instead of the positional parameters?"

Jan Schampera, 2010/07/14 19:42 ()

I hope it is understandable now, personally I never used this feature. Please provide suggestions if you like.

Thanks for the feedback.

Mark Szymanski, 2010/07/29 03:42 ()

I am sorry if I am just missing something here but, what is with the `> ampersand 2` in the echo commands?

Jan Schampera, 2010/07/29 09:55 ()

It's good practice to print error and diagnostic messages to the standard error output (`STDERR`). `foo > ampersand 2` does this.

Zenaan Harkness, 2010/11/04 14:34 ()

> It's good practice to print error and diagnostic messages to the standard error output (STDERR).

Agreed, but please don't do so for -h or –help output, which is expected and specified output and should therefore go to STDOUT.

Jan Schampera, 2010/11/04 18:12 ()

I kind of agree, yes. But I also made this mistake here and there. When help is wanted, the output of the help text is not a diagnistig or an error message.

Joshua, 2010/12/05 00:06 ()

What if there are multiple options and some require arguments while some do not? I can't seem to get it to work properly...

Ex)

#!/bin/bash

while getopts ":a:b:cde:f:g:" opt; do

```
case $opt in
  a)
    echo "-a was triggered, Parameter: $OPTARG" >&2
    ;;
  b)
    echo "-b was triggered, Parameter: $OPTARG" >&2
    ;;
  c)
    echo "-c was triggered, Parameter: $OPTARG" >&2
    ;;
  d)
    echo "-d was triggered, Parameter: $OPTARG" >&2
    ;;
  e)
    echo "-e was triggered, Parameter: $OPTARG" >&2
    ;;
  f)
    echo "-w was triggered, Parameter: $OPTARG" >&2
    ;;
  g)
    echo "-g was triggered, Parameter: $OPTARG" >&2
    ;;
```

```
  \?)
    echo "Invalid option: -$OPTARG" >&2
    exit 1
    ;;
  :)
    echo "Option -$OPTARG requires an argument." >&2
    exit 1
    ;;
esac
```

done

Here's my problem:

./hack.bash -a -b -a was triggered, Parameter: -b

Shouldn't it display that -a is missing an argument instead of taking the next option as the parameter. What am I doing wrong here?

Jan Schampera, 2010/12/05 06:29 ()

You're doing nothing wrong. It is like that, when getopts searches an argument, it takes the next one.

This is how most programs I know behave (`tar` , the text utils, ...).

Mark, 2011/01/29 19:42 ()

How do I get it so that with no arguments passed, it returns text saying "no arguments password, nothing triggered"?

Jan Schampera, 2011/01/29 19:50 ()

I'd do it by checking $# before the while/getopts loop, if applicable:

```
if (($# == 0)); then
  ...
fi
```

If you really need to check if `getopts` found something to process you could make up a variable for that check:

```
options_found=0

while getopts ":xyz" opt; do
  options_found=1
  ...
done

if ((!options_found)); then
  echo "no options found"
fi
```

saravanan subramanian, 2014/12/04 16:56 ()

Hi , I tried the above - but still when i run with ./tw.ksh -a -f "XXX" - it sets the variable options_found=1. I want it to run only when both options have arguments otherwise exit.

while getopts ":hf:F:a:" arg

do

options_found=1

```
       case $arg in
```

```
           a)
```

```
               CMD_INP=$OPTARG
```

```
               ;;
```

```
           F)
```

```
              HOSTS_FILE="$STAGEDIR/$OPTARG/hosts"
              if [ ! -f $HOSTS_FILE ]
              then
                  echo "File $HOSTS_FILE does not exist, exiting"
                  exit
              else
                  HOSTS_TO_FIX=`cat $HOSTS_FILE`
              fi
              ;;
          f)
              HOSTS_TO_FIX=$OPTARG
              ;;
          h|*)
              usage
              exit 1
              ;;
      esac
```

done

Reid, 2011/08/11 22:07 (), 2011/08/11 23:41 ()

Another method of checking whether it found anything at all is to run a separate if statement right before the while getopts call.

```
if ( ! getopts "abc:deh" opt); then
        echo "Usage: `basename $0` options (-ab) (-c value) (-
d) (-e) -h for help";
        exit $E_OPTERROR;
fi

while etopts "abc:deh" opt; do
     case $opt in
         a) do something;;
         b) do another;;
         c) var=$OPTARG;;
         ...
     esac
done
```

Mark, 2011/01/29 20:09 ()

Sweet - that work, thanks!

How do you get it to return multiple arguments on one line? eg. hello -ab returns "option a option b"?

Jan Schampera, 2011/01/29 21:16 ()

This isn't related to getopts. Just use variables or echo without newlines or such things, as you would do in such a case without getopts, too.

Andrea, 2011/05/02 14:22 ()

Hi. how can I control the double invocation of the same option? I don't want this situation: ./script -a xxx -a xxx!

Jan Schampera, 2011/05/02 15:03 ()

See the question above. Set a variable that handles this, a kind of flag that is set when the option is invoked, and checked if the option already was invoked. A kind of "shield".

```
A_WAS_SET=0
...
case
...
   a)
     if [[ $A_WAS_SET = 0 ]]; then
       A_WAS_SET=1
       # do something that handles -a
     else
       echo "Option -a already was used."
       exit 1
     fi
   ;;
esac
...
```

Andrea, 2011/05/03 13:57 ()

Thanks! It works!

Joe Wulf, 2011/06/22 20:33 ()

Joshua's example (from above @ 2010/12/05 01:06) asked about parsing multiple options, where some DO have required arguments, and some have OPTIONAL arguments. I've a script I'm enhancing. It takes a '-e' argument to EXECUTE (and '-i' for installation, '-r' for removal, etc...). The -e is stable by itself. My enhancement would be allowing an optional '-e <modifier>' so that the functionality would be appropriately conditionally modified. How do I define the getopts line to state that '-e' is a valid parsable option, and that it MIGHT have an argument??

Jan Schampera, 2011/06/23 06:42 (), 2011/06/23 06:58 ()

Hi,

try this trick. When you discover that `OPTARG` von `-c` is something beginning with a hyphen, then reset `OPTIND` and re-run getopts (`continue` the while loop).

The code is relatively small, but I hope you get the idea.

Oh, of course, this isn't perfect and needs some more robustness. It's just an example.

```
#!/bin/bash


while getopts :abc: opt; do
  case $opt in
    a)
      echo "option a"
    ;;
    b)
      echo "option b"
    ;;
    c)
      echo "option c"

      if [[ $OPTARG = -* ]]; then
        ((OPTIND--))
        continue
      fi

      echo "(c) argument $OPTARG"
    ;;
    \?)
      echo "WTF!"
      exit 1
    ;;
  esac
done
```

Reid, 2011/08/11 21:29 ()

Another method to have an "optional" argument would be to have both a lower and uppercase version of the option, with one requiring the argument and one not requiring it.

Jay, 2011/07/27 18:10 ()

Bold Text What if you have a flag with an OPTIONAL argument; say the call can either be with -a username or just -a. Defined with just a: it complains there is no argument. I want it to use the argument if there is one, else use default defined elsewhere.

Arvid Requate, 2011/10/07 10:04 ()

The builtin getopts can be used to parse long options by putting a dash character followed by a colon into the optstring ("getopts 'h-:'" or "getopts – '-:'"), here is an example how it can be done:

http://stackoverflow.com/questions/402377/using-getopts-in-bash-shell-script-to-get-long-and-short-command-line-options/7680682#7680682 (http://stackoverflow.com/questions/402377/using-getopts-in-bash-shell-script-to-get-long-and-short-command-line-options/7680682#7680682)

This approach is portable to Debian Almquist shell ("dash").

Jan Schampera, 2011/10/08 08:09 ()

Very nice trick!

Another way I could imagine (and I'll try some test code some day) is preprocessing the positional parameters and convert long options to short options before using getopts.

Abi Michael, 2012/02/06 04:06 ()

Thank you so much. Such a wonderful tutorial!!!

Jan Sidlo, 2012/03/09 22:04 ()

I've got a question. How do I get those mass-arguments or operands? When I have list of input files at the end. Like this ./script -r r_arg -np input_file_1 input_file_2, or ./script -r r_arg -n -p p_arg input_file_1 input_file_2. I don't know number of operands (input files) in advance so can't use $#.

Jan Schampera, 2012/03/10 10:08 ()

After `getopts` is done, shift all processed options away with

```
shift $((OPTIND-1))
# now do something with $@
```

chandrakanth, 2012/03/30 06:11 ()

hello all Need help!!

In my case i need the switches to be strings just not characters for ex. getopt as explained here works like filename -a <arga> -b <argb> but i wanted in filename -name <arga> -age <argb> format

if not using getopt what else can be used to implement this kind in shell scripting (sh shell)

Thanks in advance

Jan Schampera, 2012/04/07 08:09 ()

See the examples in Handling positional parameters for some non-getopts way.

Shashikant, 2012/04/10 12:43 ()

Very nice explanation. It helped me.

Thanks a lot.

stib, 2012/07/23 06:19 ()

This is my attempt to have optional arguments. If the user specifies -a -b then getopts sees -b as $OPTARG for -a so we wind $OPTIND back one, and in this case give $a a default value, otherwise $a gets the value of $OPTARG. Note that it doesn't work if the user uses the options in the form -ab, only -a -b [CODE] while getopts ":a:bc" optname; do case "$optname" in "a") echo a triggered;

```
a=$OPTARG;
if [[ "$OPTARG"=="-*" ]];
 then ((OPTIND--));
 a="DEFAULT";
fi
```

;; "b") echo "b triggered"; ;; ":") echo need a value pal; ;; esac; done; echo a=$a; [/CODE]

Rafael Rinaldi, 2012/10/07 03:31 ()

I was trying to be able to give names to my options instead of using only single characters. It was a pain to do this with getopts, I don't know if I'm missing something or what. If there's anyone with the same need, heres my workaround: http://stackoverflow.com/questions/402377/using-getopts-in-bash-shell-script-to-get-long-and-short-command-line-options/6946864#6946864 (http://stackoverflow.com/questions/402377/using-getopts-in-bash-shell-script-to-get-long-and-short-command-line-options/6946864#6946864)

Jan Schampera, 2012/10/13 09:30 ()

Hi,

as noted above, `getopts` is not made to parse GNU-style long options (which is exactly what you are trying to do).

Kevin Wyman, 2013/04/06 02:33 ()

Hey all, so I'm actually having a pretty rough time wrapping my head around getopts, case and shifting. I understand that shift the positional parameter and moves left, giving it a value of [1]. The biggest area I am struggling with is using functions with getopts/case. I have my functions and their logic defined above my getopts declaration and want to assign each switch(option) an invocation of a (or a set of) function(s). Further, I have the logic of one function to be invoked by a switch (we'll use -m) only when a seperate switch (-i) is present and successfully completes first.

Breakdown: [code] variable="$2" usage() { … } func_i() { do something with $2 … } func_m() { do something after func_i completes task … } func_S() { run script under defined configuration/permissions … } #Now start processing the positional parameters while getopts ":mSi:?" args; do case $args in

```
  i) func_i;;
  m) func_m;;
  S) func_S;;
  ?) usage;;
```

esac done shift [2] [/code]

How can I control (-m) executing only after (-i) is finished doing what the function declares to do with the given argument/operand $2?

---

[1] $N-1
[2] OPTIND-1

stella, 2013/05/07 03:20 ()

It helps a lot, Thank you very much!

Adrian Micu, 2013/08/23 13:41 ()

Hello,

Thanks for the great tutorial. I have found out that if the while getopts … sequence is used inside a function, this doesn't work anymore, so getopts must be used in the "main" part of the script. I don't know why - maybe you have an idea?

Thanks, Adrian

palash holkar, 2015/08/12 22:20 ()

Hi,

I have an script, similar to lp command: mylp filename -d printername or mylp -d printername filename

Using getopt I get printername, is there any way in getopt I can get filename?

Thanks

howto/getopts_tutorial.txt Last modified: 2018/03/21 00:07 by ffox8

This site is supported by Performing Databases - your experts for database administration

Bash Hackers Wiki

Except where otherwise noted, content on this wiki is licensed under the following license:
GNU Free Documentation License 1.3