GoLinuxCloud

MENU

Beginners guide to use getopts in bash scripts & examples

Table of Contents

In this tutorial we will learn about `getopts` in bash or shell programming language. getopts is short abbreviation for "**ge**t **t**he **opt**ion**s**" which you have supplied in the form of flags to the script. It has a very specific syntax that will seem confusing at first, but, once we've looked at it fully, it should not be too complicated for you to understand.

I have written another article which can help you write a script with multiple input arguments in a very clean manner without using `getopts`.

getopt vs getopts

`getopts` is a shell builtin which is available in both the regular Bourne shell (sh) and in Bash. It originated around 1986, as a replacement for `getopt`, which was created sometime before 1980. In contrast to `getopts`, `getopt` is not built into the shell, it is a standalone program that has been ported to many different Unix and Unix-like distributions.

The main differences between `getopts` and `getopt` are as follows:

- `getopt` does not handle empty flag arguments well; `getopts` does
- `getopts` is included in the Bourne shell and Bash; `getopt` needs to be installed separately
- `getopt` allows for the parsing of long options (`--help` instead of `-h`); `getopts` does not
- `getopts` has a simpler syntax; `getopt` is more complicated (mainly because it is an external program and not a builtin)

The getopts syntax

The `getopts` builtin (not in tcsh) parses command-line arguments, making it easier to write programs that follow the Linux argument conventions.

The syntax is:

```
getopts optstring varname [arg ...]
```

- where `optstring` is a list of the valid option letters,
- `varname` is the variable that receives the options one at a time,
- and `arg` is the optional list of parameters to be processed.
- If `arg` is not present, `getopts` processes the command-line arguments.
- If `optstring` starts with a colon (`:`), the script must take care of generating error messages; otherwise, `getopts` generates error messages.

- Each time `getopts` is called and locates an argument, it increments `OPTIND` to the index of the next option to be processed.
- If the option takes an argument, bash assigns the value of the argument to `OPTARG`

Where should we use colon in optstring?

If the first character in optstring is a colon (:, the shell variable OPTARG is set to the option character found, but no output is written to standard error; otherwise, the shell variable OPTARG is unset and a diagnostic message is written to standard error
If a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.

Example-1: Use bash getopts with single argument

In this sample script we will take single argument as an input to our script using getopts. The script currently only supports `-h` as input argument which will show the usage function

```
# cat single_arg.sh
#!/bin/bash

function usage {
        echo "./$(basename $0) -h --> shows usage"
}

# list of arguments expected in the input
optstring=":h"

while getopts ${optstring} arg; do
  case ${arg} in
    h)
      echo "showing usage!"
      usage
      ;;
    :)
      echo "$0: Must supply an argument to -$OPTARG." >&2
```

```
        exit 2
        ;;
    esac
```


Let us understand the script:

- In this version of the code, the while structure evaluates the `getopts` builtin each time control transfers to the top of the loop.
- The `getopts` builtin uses the `OPTIND` variable to keep track of the index of the argument it is to process the next time it is called.
- There is **no need** to call *shift* in this example.
- In this script, the case patterns do not start with a hyphen because the value of `arg` is just the option letter (`getopts` **strips off** the *hyphen*).
- Because you tell `getopts` which options are valid and which require arguments, it can detect errors in the command line and handle them in two ways.
- This example uses a leading colon in `optstring` to specify that you check for and handle errors in your code; when `getopts` finds an invalid option, it sets `varname` (from our syntax) to `?` and `OPTARG` to the option letter. When it finds an option that is missing an argument, `getopts` sets `varname` to `:` and `OPTARG` to the option lacking an argument.
- The `?` case pattern specifies the action to take when `getopts` detects an invalid option.

- If you omit the leading colon from `optstring`, both an invalid option and a missing option argument cause `varname` to be assigned the string
- `OPTARG` is not set and `getopts` writes its own diagnostic message to standard error.
- Generally this method is less desirable because you have less control over what the user sees when an error occurs.

When I execute the script with `-h` flag:

```
# bash single_arg.sh -h
showing usage!
./single_arg.sh -h --> shows usage
```

Again if we execute the same script with some other flag:

```
# bash single_arg.sh -m
Invalid option: -m.
```

Example-2: Collect multiple input arguments

- In this example script we will collect multiple input arguments using `getopts`.
- Our `optstring` variable contains the list of supported input arguments
- We have added a `:` (colon) in the `optstring` variable so that the script itself handles any errors
- I have also added some DEBUG output so you can understand how input arguments are processed with `getopts`

```
# cat multi_arg.sh
#!/bin/bash

function usage {
        echo "Usage: $(basename $0) [-abcd]" 2>&1
        echo '   -a   shows a in the output'
        echo '   -b   shows b in the output'
```

```
if [[ ${#} -eq 0 ]]; then
   usage
fi

# Define list of arguments expected in the input
optstring=":abcd"

while getopts ${optstring} arg; do
  case "${arg}" in
    a) echo "Option 'a' was called" ;;
    b) echo "Option 'b' was called" ;;
    c) echo "Option 'c' was called" ;;
    d) echo "Option 'd' was called" ;;
```

Here we execute the script with all the 4 supported options. The `OPTIND` value is `5` i.e.

`4 input arguments + 1 = 5`

```
# ./multi_arg.sh -a -b -c -d
Option 'a' was called
Option 'b' was called
Option 'c' was called
Option 'd' was called
All ARGS: -a -b -c -d
1st arg: -a
2nd arg: -b
3rd arg: -c
4th arg: -d
OPTIND: 5
```

If we execute the script with a wrong argument

```
# ./multi_arg.sh -f
Invalid option: -f.
```

```
    -d    shows d in the output
```

We can also combine all the input arguments and `getopts` will separate them and consider each alphabet individually

```
# ./multi_arg.sh -abcd
Option 'a' was called
Option 'b' was called
Option 'c' was called
Option 'd' was called
All ARGS: -abcd
1st arg: -abcd
2nd arg:
3rd arg:
4th arg:
OPTIND: 2
```

Although as you see, for the shell script `-abcd` was considered was single argument but `getopts` split the input argument and took individual flag as an input

Example-3: Use getopts in a shell script which will generate random password

- Now that we are familiar with the syntax and usage of `getopts` in bash or shell scripts. Let us take more practical example which will have multiple input arguments.
- In this script we will use `getopts` to collect input arguments and then using those arguments the script will generate a random password
- The script expects some input argument or else it will fail to execute
- You can use `-s` to append special character to the generated password
- Use `-l` to define the length of the password, the default length which the script will use is 48
- Use `-v` to increase the verbosity level
- We have defined additional colon after 'l' as it expects an input argument

```
function usage {
        echo "Usage: $(basename $0) [-vs] [-l LENGTH]" 2>&1
        echo 'Generate a random password.'
        echo '   -l LENGTH   Specify the password length'
        echo '   -s          Append a special character to the password.'
        echo '   -v          Increase verbosity.'
        exit 1
}

function print_out {
   local MESSAGE="${@}"
   if [[ "${VERBOSE}" == true ]];then
      echo "${MESSAGE}"
   fi
}

# Set default password length
LENGTH=48

# if no input argument found, exit the script with usage
if [[ ${#} -eq 0 ]]; then
   usage
```

Now we execute this script with `-s` to append a special character to the password. Since we did not use `-v` the output is very brief and only contains the password of 48 length

```
# ./gen_pwd.sh -s
153d82f5700bc0377c3c64808e90d32d8b3e1ef5454c8d0e)
```

We use `-v` this time for a more verbose output

```
# ./gen_pwd.sh -v
Verbose mode is ON
Generating a password
```

Next we also define a length of the password

```
# ./gen_pwd.sh -v -l30
Verbose mode is ON
Generating a password
Done
Here is your password
080bf7350785f1074bb5468f0f20c3
```

What's Next

Now that you are familiar with `getopts` I would suggest you also to learn about writing script using `case` and `while` loop for input flags

How to pass multiple parameters in shell script in Linux

Conclusion

In this tutorial we learned about `getopts` and how it is different from `getopt`. I have shared different examples with `getopts` syntax which can help absolute beginners starting with shell scripting. This can be useful for small scripts but I wouldn't recommend it for big scripts where you have to manage multiple input arguments with different types of values as it needs more control over the input flags and how you loop over individual flag.

Lastly I hope this article was helpful. So, let me know your suggestions and feedback using the comment section.

If my articles on **GoLinuxCloud** has helped you, kindly consider buying me a coffee as a token of appreciation.

For any other feedbacks or questions you can either use the comments section or contact me form.

Thank You for your support!!

5 thoughts on “Beginners guide to use getopts in bash scripts & examples”

david

try your script from example three, I can't enter a length for the password length. I gives me errors. From other tutorials I tried, I learned that the optionstring should be declared as follows:

```
optstring="svl:"
```

instead of:

```
optstring=":svl"
```

With this replacement of the colon behind the 'l', the 'l' parameter expects a value to be entered. Then you can run a command like:

```
# ./gen_pwd.sh -v -l30
```

or:

```
# ./gen_pwd.sh -v -l 30
```

Please correct me if I'm wrong, I'm not an expert (that's why I followed the tutorial in the first place)

Reply

admin

december 29, 2020 at 4:34 pm

Thanks for highlighting this, I have updated the post and also added some more information regarding the position of the colon. We can add it in the beginning and in the end based on the requirement. Here it makes sense to have it after "l" as we expect an input argument for

Reply

dieter

november 6, 2021 at 11:58 am

Very helpful, thanks a lot

Reply

rawi

march 23, 2022 at 9:19 pm

thanks for sharing
in Example 1, it's missing : at the end of optstring, it should be ":h:" otherwise this case will never be met

```
:)
      echo "$0: Must supply an argument to -$OPTARG." >&2
      exit 1
      ;;
```

Reply

admin

march 28, 2022 at 12:26 pm

To achieve this we would need two arguments, something like:

```
# list of arguments expected in the input
```

```
# sh  script.sh -a
script.sh: Must supply an argument to -a
```

Reply

Leave a Comment

Name *

Email *

Save my name and email in this browser for the next time I comment.

Notify me via e-mail if anyone answers my comment.