

Using getopts in bash shell script to get long and short command line options

I wish to have long and short forms of command line options invoked using my shell script.

I know that `getopts` can be used, but like in Perl, I have not been able to do the same with shell.

Any ideas on how this can be done, so that I can use options like:

```
./shell.sh --copyfile abc.pl /tmp/
./shell.sh -c abc.pl /tmp/
```

In the above, both the commands mean the same thing to my shell, but using `getopts`, I have not been able to implement these?

shell

unix

getopt

getopts

- 2IMHO, the accepted answer is not the best one. It does not show how to use getopts to handle both "-" and "--" arguments, which can be done, as @Arvid Requate demonstrated. I'm inserting another answer using similar concept, but also deals with user error of "forgetting" to insert values for arguments that are needed. Key point: getopts can be made to work. User should avoid using "getopt" instead if cross-platform portability is needed. Also, getopts is part of POSIX standard for shells, so it is likely to be portable. – pauljohn32 Oct 17 '17 at 14:33
- Possible duplicate of [How do I parse command line arguments in Bash?](#) – rds Oct 17 at 16:03

29 Answers

The bash `getopts` builtin does not support long option names with the double-dash prefix. It only supports single-character options.

There is a shell tool `getopt` which is another program, not a bash builtin. The GNU implementation of `getopt(3)` (used by the command-line `getopt(1)` on Linux) supports parsing long options.

But the BSD implementation of `getopt` (e.g. on Mac OS X) does not support long options.

Some other answers show a solution for using the bash builtin `getopts` to mimic long options. That solution actually makes a short option whose character is "-". So you get "--" as the flag. Then anything following that becomes OPTARG, and you test the OPTARG with a nested `case`.

This is clever, but it comes with caveats:

- `getopts` can't enforce the opt spec. It can't return errors if the user supplies an invalid option. You have to do your own error-checking as you parse OPTARG.
- OPTARG is used for the long option name, which complicates usage when your long option itself has an argument. You end up having to code that yourself as an additional case.

So while it is possible to write more code to work around the lack of support for long options, this is a lot more work and partially defeats the purpose of using a getopt parser to simplify your code.

- 5So. What is the cross-platform, portable solution? – troelskn Oct 17 '09 at 23:05
- 6GNU Getopt seems to be the only choice. On Mac, install GNU getopt from macports. On Windows, I'd install GNU getopt with Cygwin. – Bill Karwin Oct 18 '09 at 4:23
- 1Apparently, ksh getopts *can* handle long options. – Tgr Jul 23 '10 at 14:13
- 1@Bill +1, although it is also fairly simple to build getopt from source ([software.frodo.looijaard.name/getopt](#)) on Mac. You can also check the version of getopt installed on your system from within scripts with "getopt -T; echo \$?". – Chinasaur Aug 30 '11 at 17:55
- 8@Bill Karwin: "The bash getopts builtin does not support long option names with the double-dash prefix." But getopts can be made to support long options: see [stackoverflow.com/a/7680682/915044](#) below. – TomRoche Aug 11 '14 at 0:47

`getopt` and `getopts` are different beasts, and people seem to have a bit of misunderstanding of what they do. `getopts` is a built-in command to `bash` to process command-line options in a loop and assign each found option and value in turn to built-in variables, so you can further process them. `getopt`, however, is an external utility program, and it *doesn't actually process your options for you* the way that e.g. bash `getopts`, the Perl `Getopt` module or the Python `optparse` / `argparse` modules do. All that `getopt` does is canonicalize the options that are passed in — i.e. convert them to a more standard form, so that it's easier for a shell script to process them. For example, an application of `getopt` might convert the following:

```
myscript -ab infile.txt -ooutfile.txt
```

into this:

```
myscript -a -b -o outfile.txt infile.txt
```

You have to do the actual processing yourself. You don't have to use `getopt` at all if you make various restrictions on the way you can specify options:

- only put one option per argument;
- all options go before any positional parameters (i.e. non-option arguments);
- for options with values (e.g. `-o` above), the value has to go as a separate argument (after a space).

Why use `getopt` instead of `getopts`? The basic reason is that only GNU `getopt` gives you support for long-named command-line options.¹ (GNU `getopt` is the default on Linux. Mac OS X and FreeBSD come with a basic and not-very-useful `getopt`, but the GNU version can be installed; see below.)

For example, here's an example of using GNU `getopt` , from a script of mine called `javawrap` :

```
# NOTE: This requires GNU getopt.  On Mac OS X and FreeBSD, you have to install this
# separately; see below.
TEMP=`getopt -o vdm: --long verbose,debug,memory:,debugfile:,minheap:,maxheap: \
    -n 'javawrap' -- "$@"`

if [ $? != 0 ] ; then echo "Terminating..." >&2 ; exit 1 ; fi

# Note the quotes around `TEMP': they are essential!
eval set -- "$TEMP"

VERBOSE=false
DEBUG=false
MEMORY=
DEBUGFILE=
JAVA_MISC_OPT=
while true; do
    case "$1" in
        -v | --verbose ) VERBOSE=true; shift ;;
        -d | --debug ) DEBUG=true; shift ;;
        -m | --memory ) MEMORY="$2"; shift 2 ;;
        --debugfile ) DEBUGFILE="$2"; shift 2 ;;
        --minheap )
            JAVA_MISC_OPT="$JAVA_MISC_OPT -XX:MinHeapFreeRatio=$2"; shift 2 ;;
        --maxheap )
            JAVA_MISC_OPT="$JAVA_MISC_OPT -XX:MaxHeapFreeRatio=$2"; shift 2 ;;
        -- ) shift; break ;;
        * ) break ;;
    esac
done
```

This lets you specify options like `--verbose -dm4096 --minh=20 --maxhe 40 --debugfi="/Users/John Johnson/debug.txt"` or similar. The effect of the call to `getopt` is to canonicalize the options to `--verbose -d -m 4096 --minheap 20 --maxheap 40 --debugfile "/Users/John Johnson/debug.txt"` so that you can more easily process them. The quoting around `"$1"` and `"$2"` is important as it ensures that arguments with spaces in them get handled properly.

If you delete the first 9 lines (everything up through the `eval set` line), the code will *still work*! However, your code will be much pickier in what sorts of options it accepts: In particular, you'll have to specify all options in the "canonical" form described above. With the use of `getopt` , however, you can group single-letter options, use shorter non-ambiguous forms of long-options, use either the `--file foo.txt` or `--file=foo.txt` style, use either the `-m 4096` or `-m4096` style, mix options and non-options in any order, etc. `getopt` also outputs an error message if unrecognized or ambiguous options are found.


NOTE: There are actually two *totally different* versions of `getopt` , basic `getopt` and GNU `getopt` , with different features and different calling conventions.² Basic `getopt` is quite broken: Not only does it not handle long options, it also can't even handle embedded spaces inside of arguments or empty arguments, whereas `getopts` does do this right. The above code will not work in basic `getopt` . GNU `getopt` is installed by default on Linux, but on Mac OS X and FreeBSD it needs to be installed separately. On Mac OS X, install MacPorts (<http://www.macports.org>) and then do `sudo port install getopt` to install GNU `getopt` (usually into `/opt/local/bin`), and make sure that `/opt/local/bin` is in your shell path ahead of `/usr/bin` . On FreeBSD, install `misc/getopt` .

A quick guide to modifying the example code for your own program: Of the first few lines, all is "boilerplate" that should stay the same, except the line that calls `getopt` . You should change the program name after `-n` , specify short options after `-o` , and long options after `--long` . Put a colon after options that take a value.

Finally, if you see code that has just `set` instead of `eval set` , it was written for BSD `getopt` . You should change it to use the `eval set` style, which works fine with both versions of `getopt` , while the plain `set` doesn't work right with GNU `getopt` .

¹Actually, `getopts` in `ksh93` supports long-named options, but this shell isn't used as often as `bash` . In `zsh` , use `zparseopts` to get this functionality.

²Technically, "GNU `getopt` " is a misnomer; this version was actually written for Linux rather than the GNU project. However, it follows all the GNU conventions, and the term "GNU `getopt` " is commonly used (e.g. on FreeBSD).

2	This was very helpful, the idea of using getopt to check the options and then process those options in a very simple loop worked really well when I wanted to add long style options to a bash script. Thanks. – ianmjones Dec 21 '11 at 11:48
2	<code>getopt</code> on Linux is not a GNU utility and the traditional <code>getopt</code> doesn't come initially from BSD but from AT&T Unix. <code>ksh93</code> 's <code>getopts</code> (also from AT&T) supports GNU-style long options. – Stephane Chazelas Jan 29 '13 at 15:06
	@StephaneChazelas -- edited to reflect your comments. I still prefer the term "GNU getopt" even though it's a misnomer, because this version follows GNU conventions and generally acts like a GNU program (e.g. making use of <code>POSIXLY_CORRECT</code>), while "Linux-enhanced getopt" wrongly suggests that this version exists only on Linux. – Urban Vagabond Jun 24 '13 at 4:07 
1	It comes from the util-linux package, so it is Linux only as that bundle of software is meant for Linux only (that <code>getopt</code> could easily be ported to other Unices, but many other software in <code>util-linux</code> are Linux-specific). All non-GNU programs making use of GNU <code>getopt(3)</code> understand <code>\$POSIX_CORRECT</code> . For instance, you wouldn't say that <code>aplay</code> is GNU just on those grounds. I suspect that when FreeBSD mention GNU <code>getopt</code> , they mean the GNU <code>getopt(3)</code> C API. – Stephane Chazelas Jun 24 '13 at 6:39
	@StephaneChazelas -- FreeBSD has an error message "Build dependency: Please install GNU getopt" that unambiguously refers to the <code>getopt</code> util, not <code>getopt(3)</code> . – Urban Vagabond Oct 17 '13 at 22:53

The Bash builtin `getopts` function can be used to parse long options by putting a dash character followed by a colon into the `optspec`:

```
#!/usr/bin/env bash
optspec=":hv-:"
while getopts "$optspec" optchar; do
    case "${optchar}" in
        -)
            case "${OPTARG}" in
                loglevel)
                    val="${!OPTARG}"; OPTIND=$(( $OPTARG + 1 ))
                    echo "Parsing option: '--${OPTARG}', value: '${val}'" >&2;
                    ;;
            esac
        *)
            ;;
    esac
done
```

```
        loglevel=*)
            val=${OPTARG#*=}
            opt=${OPTARG%$val}
            echo "Parsing option: '--${opt}', value: '${val}'" >&2
            ;;
        *)
            if [ "$OPTERR" = 1 ] && [ "${optspec:0:1}" != ":" ]; then
                echo "Unknown option --${OPTARG}" >&2
            fi
            ;;
    esac;;
h)
    echo "usage: $0 [-v] [--loglevel[=]<value>]" >&2
    exit 2
    ;;
v)
    echo "Parsing option: '-${optchar}'" >&2
    ;;
*)
    if [ "$OPTERR" != 1 ] || [ "${optspec:0:1}" = ":" ]; then
        echo "Non-option argument: '--${OPTARG}'" >&2
    fi
    ;;
done
esac
```

After copying to executable file name= `getopts_test.sh` in the current working directory, one can produce output like

```
$ ./getopts_test.sh
$ ./getopts_test.sh -f
Non-option argument: '-f'
$ ./getopts_test.sh -h
usage: code/getopts_test.sh [-v] [--loglevel[=]<value>]
$ ./getopts_test.sh --help
$ ./getopts_test.sh -v
Parsing option: '-v'
$ ./getopts_test.sh --very-bad
$ ./getopts_test.sh --loglevel
Parsing option: '--loglevel', value: ''
$ ./getopts_test.sh --loglevel 11
Parsing option: '--loglevel', value: '11'
$ ./getopts_test.sh --loglevel=11
Parsing option: '--loglevel', value: '11'
```

Obviously `getopts` neither performs `OPTERR` checking nor option-argument parsing for the long options. The script fragment above shows how this may be done manually. The basic principle also works in the Debian Almquist shell ("dash"). Note the special case:

```
getopts -- "-:" ## without the option terminator "-- " bash complains about "-:"
getopts "-:"    ## this works in the Debian Almquist shell ("dash")
```

Note that, as GreyCat from over at <http://mywiki.woledge.org/BashFAQ> points out, this trick exploits a non-standard behaviour of the shell which permits the option-argument (i.e. the filename in "-f filename") to be concatenated to the option (as in "-ffilename"). The POSIX standard says there must be a space between them, which in the case of "-- longoption" would terminate the option-parsing and turn all longoptions into non-option arguments.

- 2
- Doesn't your 'h' case need to go above your '*' in the outer case? – Rhys Ulerich Sep 21 '12 at 20:12
- 1
- The h) case should appear before the *) one otherwise the latter catch all remaining options so the -h one is never processed. – jlliagre May 30 '14 at 9:26
- 1
- One question: what is the semantics of ! in val="\${!OPTIND} ? – TomRoche Aug 11 '14 at 0:41
- 2
- @ecbrodie: It is because two arguments have actually been processed, as opposed to just one. The first argument is the word "loglevel", and the next is the argument to *that* argument. Meanwhile, `getopts` automatically only increments `OPTIND` with 1, but in our case we need it to increment by 2, so we increment it by 1 manually, then let `getopts` increment it by 1 again for us automatically. – Victor Zamanian Oct 14 '14 at 22:42
- 2
- Since we're into bash equilibrium here: Naked variable names are allowed inside arithmetic expressions, no \$ necessary. `OPTIND=$(($OPTIND + 1))` can be just `OPTIND=$((OPTIND + 1))`. Even more interestingly, you can even assign and increase variables inside an arithmetic expression, so it's possible to abbreviate it further to `:=$((++OPTIND))`, or even `((++OPTIND))` taking into account that `++OPTIND` will always be positive, so it won't trip up a shell run with the `-e` option. :-)
gnu.org/software/bash/manual/html_node/Shell-Arithmetic.html – clacke Apr 8 '16 at 13:12

The built-in `getopts` command is still, AFAIK, limited to single-character options only.

There is (or used to be) an external program `getopt` that would reorganize a set of options such that it was easier to parse. You could adapt that design to handle long options too. Example usage:

```
aflag=no
bflag=no
flist=""
set -- $(getopt abf: "$@")
while [ $# -gt 0 ]
do
    case "$1" in
        (-a) aflag=yes;;
        (-b) bflag=yes;;
        (-f) flist="$flist $2"; shift;;
        (--) shift; break;;
        (-*) echo "$0: error - unrecognized option $1" 1>&2; exit 1;;
        (*) break;;
    esac
    shift
done
```

11/4/2018	unix - Using getopt in bash shell script to get long and short command line options - Stack Overflow
	<pre># Process remaining non-option arguments ...</pre>
	You could use a similar scheme with a <code>getoptlong</code> command.
	Note that the fundamental weakness with the external <code>getopt</code> program is the difficulty of handling arguments with spaces in them, and in preserving those spaces accurately. This is why the built-in <code>getopts</code> is superior, albeit limited by the fact it only handles single-letter options.
10	<code>getopt</code> , except for the GNU version (which has a different calling convention), is fundamentally broken. Do not use it. Please use <code>**getopts</code> instead bash-hackers.org/wiki/doku.php/howto/getopts_tutorial – hendry Aug 20 '09 at 23:21
8	@hendry - from your own link: "Note that <code>getopts</code> is not able to parse GNU-style long options (<code>--myoption</code>) or XF86-style long options (<code>-myoption</code>)!" – Tom Auger Jul 18 '11 at 19:32
1	Jonathan -- you should rewrite the example to use <code>eval set</code> with quotes (see my answer below) so that it also works correctly with GNU <code>getopt</code> (the default on Linux) and handles spaces correctly. – Urban Vagabond Jul 22 '12 at 0:25
	@UrbanVagabond: I'm not sure why I should do that. The question is tagged Unix, not Linux. I'm showing the traditional mechanism, deliberately, and it has issues with blanks in arguments, etc. You can demonstrate the modern Linux-specific version if you wish, and your answer does that. (I note, passim, that your use of <code>\${1+"\$@"}</code> is quaint and at odds with what's necessary in modern shells and specifically with any shell you'd find on Linux. See Using \$1:+"\$@" in /bin/sh for a discussion of that notation.) – Jonathan Leffler Jul 22 '12 at 1:13
	You should do it because <code>eval set</code> does the right thing with both GNU and BSD <code>getopt</code> , whereas plain <code>set</code> only does the right thing with BSD <code>getopt</code> . So you may as well use <code>eval set</code> to encourage people to get in the habit of doing this. BTW thanks, I didn't realize that <code>\${1+"\$@"}</code> wasn't needed any more. I have to write things that work both on Mac OS X and Linux -- between the two of them they force lots of portability. I just checked and <code>"\$@"</code> does indeed do the right thing on all of <code>sh</code> , <code>bash</code> , <code>ksh</code> , and <code>zsh</code> under Mac OS X; surely under Linux too. – Urban Vagabond Jul 22 '12 at 7:36

Here's an example that actually uses `getopt` with long options:

```
aflag=no
bflag=no
cargument=none

# options may be followed by one colon to indicate they have a required argument
if ! options=$(getopt -o abc: -l along,blong,clong: -- "$@")
then
    # something went wrong, getopt will put out an error message for us
    exit 1
fi

set -- $options

while [ $# -gt 0 ]
do
    case $1 in
        -a|--along) aflag="yes" ;;
        -b|--blong) bflag="yes" ;;
        # for options with required arguments, an additional shift is required
        -c|--clong) cargument="$2" ; shift;;
        --) shift; break;;
        (-*) echo "$0: error - unrecognized option $1" 1>&2; exit 1;;
        (*) break;;
    esac
    shift
done
```

3	+1 for the long option example and conditional check of <code>getopt</code> return value. – Jason McCreary Dec 9 '11 at 18:57
	You should rewrite the example to use <code>eval set</code> with quotes (see my answer below) so that it also works correctly with GNU <code>getopt</code> (the default on Linux) and handles spaces correctly. – Urban Vagabond Jul 22 '12 at 0:25
	This does work with GNU <code>getopt</code> . – Izap Jul 12 '13 at 8:47
1	Oh wait, you are right. Adding <code>eval</code> helps! Thanks. – Izap Jul 12 '13 at 8:57
	This is using <code>getopt</code> while the question is about <code>getopts</code> though. – Niklas Berglund Aug 26 '16 at 12:42

Long options can be parsed by the standard `getopts` builtin as “arguments” to the `-` “option”

This is portable and native POSIX shell – no external programs or bashisms are needed.

This guide implements long options as arguments to the `-` option, so `--alpha` is seen by `getopts` as `-` with argument `alpha` and `--bravo=foo` is seen as `-` with argument `bravo=foo`. The true argument can be harvested with a simple replacement: `${OPTARG#*=}`.

In this example, `-b` (and its long form, `--bravo`) has a mandatory option (note the manual reconstruction of enforcing that for the long form). Non-boolean options to long arguments come after equals signs, e.g. `--bravo=foo` (space delimiters for long options would be hard to implement).

Because this uses `getopts`, this solution supports usage like `cmd -ac --bravo=foo -d FILE` (which has combined options `-a` and `-c` and interleaves long options with standard options) while most other answers here either struggle or fail to do that.

```
while getopts ab:c-: arg; do
    case $arg in
        a ) ARG_A=true ;;
        b ) ARG_B="$OPTARG" ;;
        c ) ARG_C=true ;;
        - ) LONG_OPTARG="${OPTARG#*=}"
            case $OPTARG in
                alpha ) ARG_A=true ;;
            esac
    esac
done
```



```
bravo=?* ) ARG_B="$LONG_OPTARG" ;;
bravo*   ) echo "No arg for --$OPTARG option" >&2; exit 2 ;;
charlie  ) ARG_C=true ;;
alpha* | charlie* )
    echo "No arg allowed for --$OPTARG option" >&2; exit 2 ;;
'' )      break ;; # "--" terminates argument processing
* )      echo "Illegal option --$OPTARG" >&2; exit 2 ;;
esac ;;
\? ) exit 2 ;; # getopt already reported the illegal option
done
shift $((OPTIND-1)) # remove parsed options and args from $@ list
```

When the argument is a dash (-), it has two more components: the flag name and (optionally) its argument. I delimit these the standard way any command would, with the first equals sign (=). \$LONG_OPTARG is therefore merely the content of \$OPTARG without the flag name or equals sign.

The inner case implements long options manually, so it needs some housekeeping:

- bravo=? matches --bravo=foo but not --bravo= (note: case stops after the first match)
- bravo* follows, noting the missing required argument in --bravo and --bravo=
- alpha* | charlie* catches arguments given to the options that don't support them
- '' is present to support non-options that happen to start with dashes
- * catches all other long options and recreates the error thrown by getopt for an invalid option

You don't necessarily need all of those housekeeping items. For example, perhaps you want --bravo to have an *optional* argument (which -b can't support due to a limitation in getopt). Merely remove the =? and the related failure case and then call \${ARG_B:=\$DEFAULT_ARG_B} the first time you use \$ARG_B .

	Very nice self-contained solution. One question: Since letter-c needs no argument, would it not be sufficient to use letter-c) ?; the * seems redundant. – Philip Kearns Apr 14 '15 at 15:26
	@PhilipKearns was right, the * is technically unnecessary in letter-c*) , but without it, specifying arguments like --letter-c=oops will be cited as an "illegal option" rather than "unexpected option to --letter-c ." I have added a line to handle that. – Adam Katz Apr 14 '15 at 16:14
	(In making this answer more legible, I renamed option --letter-c to --charlie) – Adam Katz Apr 16 '16 at 1:40
1	@Arne Positional arguments are bad UX; they're hard to understand and optional arguments are quite messy. getopt stops at the first positional argument since it's not designed to deal with them. This allows sub-commands with their own arguments, e.g. git diff --color , so I'd interpret command --foo=moo bar --baz waz as having --foo as an argument to command and --baz waz as an argument (with option) to the bar sub-command. This can be done with the above code. I reject --bravo - blah because --bravo requires an argument and it is unclear that -blah isn't another option. – Adam Katz Sep 20 '16 at 16:09
1	@AdamKatz: I wrote a small article with this: draketo.de/english/free-software/shell-argument-parsing — includes repeated reading of remaining arguments to catch trailing options. – Arne Babenhauserheide Apr 25 '17 at 8:35

Take a look at **shFlags** which is a portable shell library (meaning: sh, bash, dash, ksh, zsh on Linux, Solaris, etc.).

It makes adding new flags as simple as adding one line to your script, and it provides an auto generated usage function.

Here is a simple Hello, world! using **shFlag**:

```
#!/bin/sh

# source shflags from current directory
. ./shflags

# define a 'name' command-line string flag
DEFINE_string 'name' 'world' 'name to say hello to' 'n'

# parse the command-line
FLAGS "$@" || exit 1
eval set -- "${FLAGS_ARGV}"

# say hello
echo "Hello, ${FLAGS_name}!"
```

For OSes that have the enhanced getopt that supports long options (e.g. Linux), you can do:

```
$ ./hello_world.sh --name Kate
Hello, Kate!
```

For the rest, you must use the short option:

```
$ ./hello_world.sh -n Kate
Hello, Kate!
```

Adding a new flag is as simple as adding a new DEFINE_ call .

1	This is fantastic but unfortunately my getopt (OS X) doesn't support spaces in arguments :/ wonder if there is an alternative. – Alastair Stuart Mar 16 '11 at 16:16
	@AlastairStuart -- there is indeed an alternative on OS X. Use MacPorts to install GNU getopt (it will usually be installed into /opt/local/bin/getopt). – Urban Vagabond Jul 9 '12 at 5:56
1	@UrbanVagabond – the installation of non system default tools is unfortunately not an acceptable requirement for a sufficiently portable tool. – Alastair Stuart Jul 9 '12 at 10:39
	@AlastairStuart – see my answer for a portable solution that uses the getopt builtin rather than GNU getopt. It's the same as basic getopt usage but with an extra iteration for long options. – Adam Katz Aug 31 '17 at 15:03

Using `getopts` with short/long options and arguments

Works with all combinations, e.G.:

- `foobar -f --bar`
- `foobar --foo -b`
- `foobar -bf --bar --foobar`
- `foobar -fbFBAshorty --bar -FB --arguments=longhorn`
- `foobar -fA "text shorty" -B --arguments="text longhorn"`
- `bash foobar -F --barfoo`
- `sh foobar -B --foobar - ...`
- `bash ./foobar -F --bar`

Some declarations for these example

```
Options=$@
Optnum=##
sfoo='no '
sbar='no '
sfoobar='no '
sbarfoo='no '
sarguments='no '
sARG=empty
lfoo='no '
lbar='no '
lfoobar='no '
lbarfoo='no '
larguments='no '
lARG=empty
```

How the Usage function would look

```
function _usage()
{
    ##### U S A G E : Help and ERROR #####
    cat <<EOF
    foobar $Options
    $*

        Usage: foobar <[options]>
    Options:
        -b  --bar           Set bar to yes      ($foo)
        -f  --foo           Set foo to yes       ($bart)
        -h  --help          Show this message
        -A  --arguments=... Set arguments to yes ($arguments) AND get
ARGUMENT ($ARG)
        -B  --barfoo        Set barfoo to yes   ($barfoo)
        -F  --foobar        Set foobar to yes    ($foobar)

    EOF
}

[ $# = 0 ] && _usage " >>>>>>> no options given "
```

`getopts` with long/short flags as well as long arguments

```
while getopts 'bfh-A:BF' OPTION ; do
    case "$OPTION" in
        b ) sbar=yes           ;;
        f ) sfoo=yes           ;;
        h ) _usage              ;;
        A ) sarguments=yes;sARG="$OPTARG" ;;
        B ) sbarfoo=yes        ;;
        F ) sfoobar=yes         ;;
        - ) [ $OPTIND -ge 1 ] && optind=$(expr $OPTIND - 1 ) || optind=$OPTIND
            eval OPTION="\${$optind}"
            OPTARG=$(echo $OPTION | cut -d'=' -f2)
            OPTION=$(echo $OPTION | cut -d'=' -f1)
            case $OPTION in
                --foo          ) lfoo=yes           ;;
                --bar          ) lbar=yes           ;;
                --foobar       ) lfoobar=yes        ;;
                --barfoo       ) lbarfoo=yes        ;;
                --help         ) _usage              ;;
                --arguments    ) larguments=yes;lARG="$OPTARG" ;;
                * ) _usage " Long: >>>>>>> invalid options (long) " ;;
            esac
            OPTIND=1
            shift
        ;;
        ? ) _usage "Short: >>>>>>> invalid options (short) " ;;
    esac
done
```

Output

```
#####
echo "-----"
echo "RESULT short-foo      : $sfoo                long-foo      :
$lfoo"
echo "RESULT short-bar      : $sbar                long-bar      :
$lbar"
echo "RESULT short-foobar   : $sfoobar             long-foobar   :
$lfoobar"
echo "RESULT short-barfoo   : $sbarfoo             long-barfoo   :
$lbarfoo"
echo "RESULT short-arguments: $sarguments  with Argument = \"$sARG\"  long-arguments:
$larguments and $lARG"
```

Combining the above into a cohesive script

```
#!/bin/bash
# foobar: getopt with short and Long options AND arguments

function _cleanup ()
{
    unset -f _usage _cleanup ; return 0
}

## Clear out nested functions on exit
trap _cleanup INT EXIT RETURN

##### some declarations for these example #####
Options=@
Optnum=#
sfoo='no '
sbar='no '
sfoobar='no '
sbarfoo='no '
sarguments='no '
sARG=empty
lfoo='no '
lbar='no '
lfoobar='no '
lbarfoo='no '
larguments='no '
lARG=empty

function _usage()
{
    ##### U S A G E : Help and ERROR #####
    cat <<EOF
    foobar $Options
    $*

        Usage: foobar <[options]>
    Options:
        -b  --bar          Set bar to yes      ($foo)
        -f  --foo          Set foo to yes       ($bart)
        -h  --help         Show this message
        -A  --arguments=... Set arguments to yes ($arguments) AND get
ARGUMENT ($ARG)
        -B  --barfoo       Set barfoo to yes ($barfoo)
        -F  --foobar       Set foobar to yes ($foobar)

    EOF
}

[ $# = 0 ] && _usage " >>>>>> no options given "

#####
##### "getopts" with: short options AND Long options #####
##### AND short/Long arguments #####
while getopts ':bfh-A:BF' OPTION ; do
    case "$OPTION" in
        b ) sbar=yes                ;;
        f ) sfoo=yes                 ;;
        h ) _usage                   ;;
        A ) sarguments=yes;sARG="$OPTARG" ;;
        B ) sbarfoo=yes              ;;
        F ) sfoobar=yes              ;;
        - ) [ $OPTIND -ge 1 ] && optind=$(expr $OPTIND - 1 ) || optind=$OPTIND
            eval OPTION="\${$optind}"
            OPTARG=$(echo $OPTION | cut -d=' ' -f2)
            OPTION=$(echo $OPTION | cut -d=' ' -f1)
            case $OPTION in
                --foo      ) lfoo=yes                ;;
                --bar      ) lbar=yes                ;;
                --foobar    ) lfoobar=yes             ;;
                --barfoo    ) lbarfoo=yes             ;;
                --help      ) _usage                   ;;
                --arguments ) larguments=yes;lARG="$OPTARG" ;;
                * ) _usage " Long: >>>>>> invalid options (long) " ;;
            esac
            OPTIND=1
            shift
        ;;
        ? ) _usage "Short: >>>>>> invalid options (short) " ;;
    esac
done
```

Another way...

```
# translate long options to short
for arg
do
    delim=""
    case "$arg" in
        --help) args="${args}-h ";;
        --verbose) args="${args}-v ";;
        --config) args="${args}-c ";;
        # pass through anything else
        *) [[ "${arg:0:1}" == "-" ]] || delim="\"
            args="${args}${delim}${arg}${delim} ";;
    esac
done
# reset the translated args
eval set -- $args
# now we can process with getopt
while getopt ":hvc:" opt; do
    case $opt in
        h) usage ;;
        v) VERBOSE=true ;;
        c) source $OPTARG ;;
        \?) usage ;;
        :)
            echo "option -$OPTARG requires an argument"
            usage
            ;;
    esac
done
```

- 2
- This is by far the best solution. Very simple and does not require the dreaded "getopt". Thank you! – Caetano Sauer Dec 15 '13 at 12:19
- Doesn't this need a space in each `$args` reassignment? This could even be done with out bashisms, but this code will lose spaces in options and arguments (I don't think the `$delim` trick will work). You can instead run `set` *inside* the `for` loop if you are careful enough to empty it on only the first iteration. [Here is a safer version](#) without bashisms. – Adam Katz Apr 16 '16 at 2:06

I kind of solved this way:

```
# A string with command options
options=$@

# An array with all the arguments
arguments=( $options )

# Loop index
index=0

for argument in $options
do
    # Incrementing index
    index=`expr $index + 1`

    # The conditions
    case $argument in
        -a) echo "key $argument value ${arguments[index]}" ;;
        -abc) echo "key $argument value ${arguments[index]}" ;;
    esac
done

exit;
```

Am I being dumb or something? `getopt` and `getopts` are so confusing.

- 1
- This seems to work for me, I don't know what the problem is with this method, but it seems simple, so there must be a reason everyone else is not using it. – Billy Moon Aug 7 '11 at 10:46
- 1
- @Billy Yes, this is simple because I don't use any script to manage my parameters and etc. Basically I convert the arguments string (`$@`) to an array and I loop through it. In the loop, current value will be the key and the next one will be the value. Simple as that. – user339827 Aug 23 '11 at 23:23
- 1
- @Theodore I'm glad this was helpful to you! It was a pain to me as well. If you're interested, you can see an example of it in action here: [raw.github.com/rafaelrinaldi/swf-to-html/master/swf-to-html.sh](https://raw.githubusercontent.com/rafaelrinaldi/swf-to-html/master/swf-to-html.sh) – user339827 Aug 23 '11 at 23:24
- 2
- Definitely the easiest way I've seen. I changed it a bit such as using `i=$(($i + 1))` instead of `expr` but the concept is air-tight. – Thomas Dignan Nov 14 '12 at 7:58
- 5
- You are not dumb at all, but you may be missing a feature: `getopt(s)` can recognise options that are mixed (ex: `-ltr` or `-lt -r` as well as `-l -t -r`). And it also provides some error handling, and an easy way to shift the treated parameters away once options treatment is finished. – Olivier Dulac Nov 28 '12 at 18:08

In case you don't want the `getopt` dependency, you can do this:

```
while test $# -gt 0
do
    case $1 in

        # Normal option processing
        -h | --help)
            # usage and help
            ;;
```



```
-v | --version)
    # version info
    ;;
# ...

# Special cases
--)
    break
    ;;
--*)
    # error unknown (Long) option $1
    ;;
-?)
    # error unknown (short) option $1
    ;;

# FUN STUFF HERE:
# Split apart combined short options
-*)
    split=$1
    shift
    set -- $(echo "$split" | cut -c 2- | sed 's/./-& /g') "$@"
    continue
    ;;

# Done with options
*)
    break
    ;;
esac

# for testing purposes:
echo "$1"

shift
done
```

Of course, then you can't use long style options with one dash. And if you want to add shortened versions (e.g. --verbos instead of --verbose), then you need to add those manually.

But if you are looking to get `getopts` functionality along with long options, this is a simple way to do it.

I also put this snippet in a [gist](#).

- This seems to only work with one long option at a time, but it met my need. Thank you! – kingjeffrey Jan 21 '13 at 19:48
- In the special case `--)` there seems to be a `shift ;` missing. At the moment the `--` will stay as first non option argument. – dgw Jan 31 '14 at 15:06
- I think that this is actually the better answer, though as dgw points out the `--` option needs a `shift` in there. I say this is better because the alternatives are either platform dependent versions of `getopt` or `getopts_long` or you have to force short-options to be used only at the start of the command (i.e - you use `getopts` then process long options afterwards), whereas this gives any order and complete control. – Haravikk Feb 19 '14 at 14:18
- This answer makes me wonder why we have a thread of dozens of answers to do the job that can be done with nothing more than this *absolutely clear and straightforward* solution, and if there's any reason for the billion `getopt`(s) use cases other than proving oneself. – Florian Heigl Aug 7 '17 at 0:50

The built-in `getopts` can't do this. There is an external *getopt*(1) program that can do this, but you only get it on Linux from the *util-linux* package. It comes with an example script *getopt-parse.bash*.

There is also a [getopts_long](#) written as a shell function.

answered Dec 31 '08 at 6:28



Nietzsche-jou

11.5k 4 27 41

- 1 Does `getopts_long` (above) works on Mac OS X 10.6+ ? – David Andreoletti Jan 16 '12 at 1:57
- 3 The `getopt` was included in FreeBSD version 1.0 in 1993, and has been part of FreeBSD since then. As such, it was adopted from FreeBSD 4.x for inclusion in Apple's Darwin project. As of OS X 10.6.8, the man page included by Apple remains an exact duplicate of the FreeBSD man page. So yes, it's included in OS X and gobs of other operating systems beside Linux. -1 on this answer for the misinformation. – ghoti Mar 27 '12 at 3:41

```
#!/bin/bash
while getopts "abc:d:" flag
do
    case $flag in
        a) echo "[getopts:$OPTIND]==> -$flag";;
        b) echo "[getopts:$OPTIND]==> -$flag";;
        c) echo "[getopts:$OPTIND]==> -$flag $OPTARG";;
        d) echo "[getopts:$OPTIND]==> -$flag $OPTARG";;
    esac
done

shift $((OPTIND-1))
echo "[otheropts]==> $@"

exit
```

```
#!/bin/bash
until [ -z "$1" ]; do
  case $1 in
    "--dlong")
      shift
      if [ "${1:1:0}" != "-" ]
      then
        echo "==> dlong $1"
        shift
      fi;;
    *) echo "==> other $1"; shift;;
  esac
done
exit
```

An explanation would be nice. The first script accepts short options only while the second script has a bug in its long option argument parsing; its variable should be `"${1:0:1}"` for argument #1, substring at index 0, length 1. This does not permit mixing short and long options. – Adam Katz Jun 24 '16 at 0:05

In ksh93 , getopt does support long names...

```
while getopt "f(file):s(server):" flag
do
  echo "$flag" $OPTIND $OPTARG
done
```

Or so the tutorials I have found have said. Try it and see.

- 3 This is ksh93's getopt builtin. Apart from this syntax, it also has a more complicated syntax that also allows long options without a short equivalent, and more. – jilles Oct 9 '11 at 12:28
- 1 A reasonable answer. The OP didn't specify WHAT shell. – ghoti Mar 27 '12 at 3:44

Inventing yet another version of the wheel...

This function is a (hopefully) POSIX-compatible plain bourne shell replacement for GNU getopt. It supports short/long options which can accept mandatory/optional/no arguments, and the way in which options are specified is almost identical to GNU getopt, so conversion is trivial.

Of course this is still a sizeable chunk of code to drop into a script, but it's about half the lines of the well-known getopt_long shell function, and might be preferable in cases where you just want to replace existing GNU getopt uses.

This is pretty new code, so YMMV (and definitely please let me know if this isn't actually POSIX-compatible for any reason -- portability was the intention from the outset, but I don't have a useful POSIX test environment).

Code and example usage follows:

```
#!/bin/sh
# posix_getopt shell function
# Author: Phil S.
# Version: 1.0
# Created: 2016-07-05
# URL: http://stackoverflow.com/a/37087374/324105

# POSIX-compatible argument quoting and parameter save/restore
# http://www.etalabs.net/sh_tricks.html
# Usage:
# parameters=$(save "$@") # save the original parameters.
# eval "set -- ${parameters}" # restore the saved parameters.
save () {
  local param
  for param; do
    printf %s\\n "$param" \
      | sed "s/'/'\\\\\\\\'/g;1s/^/'/;\\$s/\\$/' \\\\\\\\"
  done
  printf %s\\n " "
}

# Exit with status $1 after displaying error message $2.
exiterr () {
  printf %s\\n "$2" >&2
  exit $1
}

# POSIX-compatible command line option parsing.
# This function supports long options and optional arguments, and is
# a (largely-compatible) drop-in replacement for GNU getopt.
#
# Instead of:
# opts=$(getopt -o "$shortopts" -l "$longopts" -- "$@")
# eval set -- ${opts}
#
# We instead use:
# opts=$(posix_getopt "$shortopts" "$longopts" "$@")
# eval "set -- ${opts}"
posix_getopt () { # args: "$shortopts" "$longopts" "$@"
  local shortopts longopts \
    arg argtype getopt nonopt opt optchar optword suffix

  shortopts="$1"
```

```

longopts="$2"
shift 2

getopt=
nonopt=
while [ $# -gt 0 ]; do
    opt=
    arg=
    argtype=
    case "$1" in
        # '--' means don't parse the remaining options
        ( -- ) {
            getopt="${getopt}$(save "$@")"
            shift $#
            break
        };;
        # process short option
        ( -[!-]* ) {
            # -x[foo]
            suffix=${1#-?} # foo
            opt=${1%$suffix} # -x
            optchar=${opt#-} # x
            case "${shortopts}" in
                ( *${optchar}::* ) { # optional argument
                    argtype=optional
                    arg="${suffix}"
                    shift
                };;
                ( *${optchar}:* ) { # required argument
                    argtype=required
                    if [ -n "${suffix}" ]; then
                        arg="${suffix}"
                        shift
                    else
                        case "$2" in
                            ( -* ) exiterr 1 "$1 requires an argument";;
                            ( ?* ) arg="$2"; shift 2;;
                            ( * ) exiterr 1 "$1 requires an argument";;
                        esac
                    fi
                };;
                ( *${optchar}* ) { # no argument
                    argtype=none
                    arg=
                    shift
                    # Handle multiple no-argument parameters combined as
                    # -xyz instead of -x -y -z. If we have just shifted
                    # parameter -xyz, we now replace it with -yz (which
                    # will be processed in the next iteration).
                    if [ -n "${suffix}" ]; then
                        eval "set -- $(save "-${suffix}")$(save "$@")"
                    fi
                };;
                ( * ) exiterr 1 "Unknown option $1";;
            esac
        };;
        # process long option
        ( --?* ) {
            # --xarg[=foo]
            suffix=${1#*=} # foo (unless there was no =)
            if [ "${suffix}" = "$1" ]; then
                suffix=
            fi
            opt=${1%$suffix} # --xarg
            optword=${opt#--} # xarg
            case ",${longopts}," in
                ( *,${optword}::,* ) { # optional argument
                    argtype=optional
                    arg="${suffix}"
                    shift
                };;
                ( *,${optword}:,* ) { # required argument
                    argtype=required
                    if [ -n "${suffix}" ]; then
                        arg="${suffix}"
                        shift
                    else
                        case "$2" in
                            ( -* ) exiterr 1 \
                                "--${optword} requires an argument";;
                            ( ?* ) arg="$2"; shift 2;;
                            ( * ) exiterr 1 \
                                "--${optword} requires an argument";;
                        esac
                    fi
                };;
                ( *,${optword},* ) { # no argument
                    if [ -n "${suffix}" ]; then
                        exiterr 1 "--${optword} does not take an argument"
                    fi
                    argtype=none
                    arg=
                    shift
                };;
                ( * ) exiterr 1 "Unknown option $1";;
            esac
        };;
        # any other parameters starting with -
        ( -* ) exiterr 1 "Unknown option $1";;
        # remember non-option parameters
        ( * ) nonopt="${nonopt}$(save "$1")"; shift;;
    esac
done

```

```

    if [ -n "${opt}" ]; then
        getopt="${getopt}$(save "$opt")"
        case "${argtype}" in
            ( optional|required ) {
                getopt="${getopt}$(save "$arg")"
            };
        esac
    fi
done

# Generate function output, suitable for:
# eval "set -- $(posix_getopt ...)"
printf %s "${getopt}"
if [ -n "${nonopt}" ]; then
    printf %s "$(save "--")${nonopt}"
fi
}

```

Example usage:

```

# Process command line options
shortopts="hvd:c::s::L:D"
longopts="help,version,directory:,client::,server::,load:,delete"
#opts=$(getopt -o "$shortopts" -l "$longopts" -n "$(basename $0)" -- "$@")
opts=$(posix_getopt "$shortopts" "$longopts" "$@")
if [ $? -eq 0 ]; then
    #eval set -- ${opts}
    eval "set -- ${opts}"
    while [ $# -gt 0 ]; do
        case "$1" in
            ( -- ) shift; break;;
            ( -h|--help ) help=1; shift; break;;
            ( -v|--version ) version_help=1; shift; break;;
            ( -d|--directory ) dir=$2; shift 2;;
            ( -c|--client ) useclient=1; client=$2; shift 2;;
            ( -s|--server ) startserver=1; server_name=$2; shift 2;;
            ( -L|--load ) load=$2; shift 2;;
            ( -D|--delete ) delete=1; shift;;
        esac
    done
else
    shorthelp=1 # getopt returned (and reported) an error.
fi

```

Here you can find a few different approaches for complex option parsing in bash: <http://mywiki.woledge.org/ComplexOptionParsing>

I did create the following one, and I think it's a good one, because it's minimal code and both long and short options work. A long option can also have multiple arguments with this approach.

```

#!/bin/bash
# Uses bash extensions.  Not portable as written.

declare -A longoptspec
longoptspec=( [loglevel]=1 ) #use associative array to declare how many arguments a Long option expects, in this case we declare that loglevel expects/has
one argument, Long options that aren't listed in this way will have zero arguments by default
optspec="h-:"
while getopts "$optspec" opt; do
while true; do
    case "${opt}" in
        -) #OPTARG is name-of-Long-option or name-of-Long-option=value
            if [[ "${OPTARG}" =~ .*=.* ]] #with this --key=value format only one argument is possible
            then
                opt=${OPTARG/=*/}
                OPTARG=${OPTARG#*=}
                ((OPTIND--))
            else #with this --key value1 value2 format multiple arguments are possible
                opt="${OPTARG}"
                OPTARG=(${@:OPTIND:${(longoptspec[$opt])}})
            fi
            ((OPTIND+=longoptspec[$opt]))
            continue #now that opt/OPTARG are set we can process them as if getopts would've given us Long options
            ;;
        loglevel)
            loglevel=$OPTARG
            ;;
        h|help)
            echo "usage: $0 [--loglevel[=]<value>]" >&2
            exit 2
            ;;
    esac
break; done
done

# End of file

```

I only write shell scripts now and then and fall out of practice, so any feedback is appreciated.

Using the strategy proposed by @Arvid Requate, we noticed some user errors. A user who forgets to include a value will accidentally have the next option's name treated as a value:

```
./getopts_test.sh --loglevel= --toc=TRUE
```

will cause the value of "loglevel" to be seen as "--toc=TRUE". This can be avoided.

I adapted some ideas about checking user error for CLI from <http://mwiki.woolledge.org/BashFAQ/035> discussion of manual parsing. I inserted error checking into handling both "-" and "--" arguments.

Then I started fiddling around with the syntax, so any errors in here are strictly my fault, not the original authors.

My approach helps users who prefer to enter long with or without the equal sign. That is, it should have same response to "--loglevel 9" as "--loglevel=9". In the --/space method, it is not possible to know for sure if the user forgets an argument, so some guessing is needed.

1. if the user has the long/equal sign format (--opt=), then a space after = triggers an error because an argument was not supplied.
2. if user has long/space arguments (--opt), this script causes a fail if no argument follows (end of command) or if argument begins with dash)

In case you are starting out on this, there is an interesting difference between "--opt=value" and "--opt value" formats. With the equal sign, the command line argument is seen as "opt=value" and the work to handle that is string parsing, to separate at the "=". In contrast, with "--opt value", the name of the argument is "opt" and we have the challenge of getting the next value supplied in the command line. That's where @Arvid Requate used \${!OPTIND}, the indirect reference. I still don't understand that, well, at all, and comments in BashFAQ seem to warn against that style (<http://mywiki.woolledge.org/BashFAQ/006>). BTW, I don't think previous poster's comments about importance of OPTIND=\$((\$OPTIND + 1)) are correct. I mean to say, I see no harm from omitting it.

In newest version of this script, flag -v means VERBOSE printout.

Save it in a file called "cli-5.sh", make executable, and any of these will work, or fail in the desired way

```
./cli-5.sh -v --loglevel=44 --toc TRUE
./cli-5.sh -v --loglevel=44 --toc=TRUE
./cli-5.sh --loglevel 7
./cli-5.sh --loglevel=8
./cli-5.sh -l9

./cli-5.sh --toc FALSE --loglevel=77
./cli-5.sh --toc=FALSE --loglevel=77

./cli-5.sh -l99 -t yyy
./cli-5.sh -l 99 -t yyy
```

Here is example output of the error-checking on user intpu

```
$ ./cli-5.sh --toc --loglevel=77
ERROR: toc value must not have dash at beginning
$ ./cli-5.sh --toc= --loglevel=77
ERROR: value for toc undefined
```

You should consider turning on -v, because it prints out internals of OPTIND and OPTARG

```
#!/usr/bin/env bash

## Paul Johnson
## 20171016
##

## Combines ideas from
## https://stackoverflow.com/questions/402377/using-getopts-in-bash-shell-script-to-get-
## long-and-short-command-line-options
## by @Arvid Requate, and http://mwiki.woolledge.org/BashFAQ/035

# What I don't understand yet:
# In @Arvid REquate's answer, we have
# val="${!OPTIND}"; OPTIND=$(( $OPTIND + 1 ))
# this works, but I don't understand it!

die() {
    printf '%s\n' "$1" >&2
    exit 1
}

printparse(){
    if [ ${VERBOSE} -gt 0 ]; then
        printf 'Parse: %s%s%s\n' "$1" "$2" "$3" >&2;
    fi
}

showme(){
    if [ ${VERBOSE} -gt 0 ]; then
        printf 'VERBOSE: %s\n' "$1" >&2;
    fi
}

VERBOSE=0
loglevel=0
toc="TRUE"

optspec=":vhl:t:-:"
while getopts "$optspec" OPTCHAR; do

    showme "OPTARG:  ${OPTARG[*]}"
    showme "OPTIND:  ${OPTIND[*]}"
    case "${OPTCHAR}" in
        -)
            case "${OPTARG}" in
                loglevel) #argument has no equal sign
```



```

    opt=${OPTARG}
    val="${!OPTIND}"
    ## check value. If negative, assume user forgot value
    showme "OPTIND is ${OPTARG} ${!OPTIND} has value \"${!OPTIND}\""
    if [[ "$val" == -* ]]; then
        die "ERROR: $opt value must not have dash at beginning"
    fi
    ## OPTIND=$(( $OPTIND + 1 )) # CAUTION! no effect?
    printparse "--${OPTARG}" " " "${val}"
    loglevel="${val}"
    shift
    ;;
loglevel=*) #argument has equal sign
    opt=${OPTARG%=*}
    val=${OPTARG#*=}
    if [ "${OPTARG#*=}" ]; then
        printparse "--$opt" "=" "${val}"
        loglevel="${val}"
        ## shift CAUTION don't shift this, fails otherwise
    else
        die "ERROR: $opt value must be supplied"
    fi
    ;;
toc) #argument has no equal sign
    opt=${OPTARG}
    val="${!OPTIND}"
    ## check value. If negative, assume user forgot value
    showme "OPTIND is ${OPTARG} ${!OPTIND} has value \"${!OPTIND}\""
    if [[ "$val" == -* ]]; then
        die "ERROR: $opt value must not have dash at beginning"
    fi
    ## OPTIND=$(( $OPTIND + 1 )) #??
    printparse "--$opt" " " "${val}"
    toc="${val}"
    shift
    ;;
toc=*) #argument has equal sign
    opt=${OPTARG%=*}
    val=${OPTARG#*=}
    if [ "${OPTARG#*=}" ]; then
        toc=${val}
        printparse "--$opt" "->" "$toc"
        ##shift ## NO! dont shift this
    else
        die "ERROR: value for $opt undefined"
    fi
    ;;

help)
    echo "usage: $0 [-v] [--loglevel[=]<value>] [--toc[=]<TRUE,FALSE>]"

    exit 2
    ;;

*)
    if [ "$OPTERR" = 1 ] && [ "${optspec:0:1}" != ":" ]; then
        echo "Unknown option --${OPTARG}" >&2
    fi
    ;;

esac;;
h|- \?|--help)
    ## must rewrite this for all of the arguments
    echo "usage: $0 [-v] [--loglevel[=]<value>] [--toc[=]<TRUE,FALSE>]" >&2
    exit 2
    ;;

l)
    loglevel=${OPTARG}
    printparse "-l" " " "${loglevel}"
    ;;

t)
    toc=${OPTARG}
    ;;


v)
    VERBOSE=1
    ;;


*)
    if [ "$OPTERR" != 1 ] || [ "${optspec:0:1}" = ":" ]; then
        echo "Non-option argument: '-${OPTARG}'" >&2
    fi
    ;;

esac
done

echo "
After Parsing values
"
echo "loglevel $loglevel"
echo "toc $toc"

```

OPTIND=\$((\$OPTIND + 1)) : it is needed whenever you 'gobble' the OPTIND's parameter (for ex: when one used --toc value : value is in parameter number \$OPTIND . Once you retrieve it for toc's value, you should tell getopt that the next parameter to parse is not value, but the one after it (hence the : OPTIND=\$((\$OPTIND + 1)) . and your script (as well as the script you refer to) are missing, after the done : shift \$((\$OPTIND -1)) (as getopt exited after parsing parameters 1 to OPTIND-1, you need to shift them out so \$@ is now any remaining "non-options" parameters – Olivier Dulac Apr 13 at 8:44 )

oh, as you shift yourself, you "shift" the parameters underneath getopt, so OPTIND is always pointing the right thing... but I find it very confusing. I believe (can't test your script right now) that you still need the shift \$((\$OPTIND - 1)) after the getopt while loop, so that \$1 now doesn't point to the original \$1 (an option) but to the first of the remaining arguments (the ones coming after all options and their values). ex: myrm -foo -bar=baz thisarg thenthisone thenanother – Olivier Dulac Apr 13 at 8:52 

I have been working on that subject for quite a long time... and made my own library which you will need to source in your main script. See [libopt4shell](#) and [cd2mpc](#) for an example. Hope it helps !

answered Mar 11 '11 at 10:59

 liealgebra

21 1

An improved solution:

```
# translate long options to short
# Note: This enable long options but disable "--?*\" in $OPTARG, or disable Long options
after "--\" in option fields.
for ((i=1; $#; i++)) ; do
    case "$1" in
        --)
            # [ ${args[${i-1}]} == ... ] || EndOpt=1 ;; & # DIRTY: we still can handle
some executions...
            EndOpt=1 ;; &
            --version) ((EndOpt)) && args[i]="1" || args[i]="-V";;
            # default case : short option use the first char of the Long option:
            --?*) ((EndOpt)) && args[i]="1" || args[i]="-${1:2:1}";;
            # pass through anything else:
            *) args[i]="1" ;;
        esac
    shift
done
# reset the translated args
set -- "${args[@]}"

function usage {
    echo "Usage: $0 [options] files" >&2
    exit $1
}

# now we can process with getopt
while getopt ":hvVc:" opt; do
    case $opt in
        h) usage ;;
        v) VERBOSE=true ;;
        V) echo $Version ; exit ;;
        c) source $OPTARG ;;
        \?) echo "unrecognized option: -$opt" ; usage -1 ;;
        :)
            echo "option -$OPTARG requires an argument"
            usage -1
            ;;
    esac
done

shift $((OPTIND-1))
[[ "$1" == "--" ]] && shift
```

Maybe it's simpler to use ksh, just for the getopt part, if need long command line options, as it can be easier done there.

```
# Working Getopts Long => KSH

#!/bin/ksh
# Getopts Long
USAGE="s(showconfig)"
USAGE+="c:(createdb)"
USAGE+="l:(createlistener)"
USAGE+="g:(generatescripts)"
USAGE+="r:(removedb)"
USAGE+="x:(removelistener)"
USAGE+="t:(createtemplate)"
USAGE+="h(help)"

while getopt "$USAGE" optchar ; do
    case $optchar in
        s) echo "Displaying Configuration" ;;
        c) echo "Creating Database $OPTARG" ;;
        l) echo "Creating Listener LISTENER_$OPTARG" ;;
        g) echo "Generating Scripts for Database $OPTARG" ;;
        r) echo "Removing Database $OPTARG" ;;
        x) echo "Removing Listener LISTENER_$OPTARG" ;;
        t) echo "Creating Database Template" ;;
        h) echo "Help" ;;
    esac
done
```

+1 -- Note that this is limited to ksh93 - from the open source AST project (AT&T Research). – Henk Langeveld Aug 16 '12 at 14:36

I wanted something without external dependencies, with strict bash support (-u), and I needed it to work on even the older bash versions. This handles various types of params:

- short bools (-h)
- short options (-i "image.jpg")
- long bools (--help)
- equals options (--file="filename.ext")
- space options (--file "filename.ext")
- concatenated bools (-hvm)

Just insert the following at the top of your script:

```
# Check if a list of params contains a specific param
# usage: if _param_variant "h|?|help p|path f|file long-thing t|test-thing" "file" ; then
...
# the global variable $key is updated to the Long notation (last entry in the pipe
delineated list, if applicable)
_param_variant() {
    for param in $1 ; do
        local variants=${param//\|/ }
        for variant in $variants ; do
            if [[ "$variant" = "$2" ]] ; then
                # Update the key to match the Long version
                local arr=(${param//\|/ })
                let last=${#arr[@]}-1
                key="${arr[$last]}"
                return 0
            fi
        done
    done
    return 1
}

# Get input parameters in short or Long notation, with no dependencies beyond bash
# usage:
#     # First, set your defaults
#     param_help=false
#     param_path="."
#     param_file=false
#     param_image=false
#     param_image_lossy=true
#     # Define allowed parameters
#     allowed_params="h|?|help p|path f|file i|image image-lossy"
#     # Get parameters from the arguments provided
#     _get_params $*
#
# Parameters will be converted into safe variable names like:
#     param_help,
#     param_path,
#     param_file,
#     param_image,
#     param_image_lossy
#
# Parameters without a value like "-h" or "--help" will be treated as
# boolean, and will be set as param_help=true
#
# Parameters can accept values in the various typical ways:
#     -i "path/goes/here"
#     --image "path/goes/here"
#     --image="path/goes/here"
#     --image=path/goes/here
# These would all result in effectively the same thing:
#     param_image="path/goes/here"
#
# Concatinated short parameters (boolean) are also supported
#     -vhm is the same as -v -h -m
_get_params(){

    local param_pair
    local key
    local value
    local shift_count

    while : ; do
        # Ensure we have a valid param. Allows this to work even in -u mode.
        if [[ $# == 0 || -z $1 ]] ; then
            break
        fi

        # Split the argument if it contains "="
        param_pair=(${1//=/ })
        # Remove preceeding dashes
        key="${param_pair[0]#--}"

        # Check for concatenated boolean short parameters.
        local nodash="${key#-}"
        local breakout=false
        if [[ "$nodash" != "$key" && ${#nodash} -gt 1 ]]; then
            # Extrapolate multiple boolean keys in single dash notation. ie. "-vmh" should
            translate to: "-v -m -h"
            local short_param_count=${#nodash}
            let new_arg_count=$#+$short_param_count-1
            local new_args=""
            # $str_pos is the current position in the short param string $nodash
            for (( str_pos=0; str_pos<new_arg_count; str_pos++ )); do
                # The first character becomes the current key
                if [ $str_pos -eq 0 ] ; then
                    key="${nodash:$str_pos:1}"
                    breakout=true
                fi
            done
        fi
        if [ "$key" = "$key" ] ; then
            value=""
        else
            value="${param_pair[1]}"
        fi
        if [ "$key" = "$key" ] ; then
            key="${key}_$shift_count"
        fi
        eval "$key=\$value"
        shift
        shift_count=$((shift_count+1))
    done
}
```

```
fi
# $arg_pos is the current position in the constructed arguments list
let arg_pos=$str_pos+1
if [ $arg_pos -gt $short_param_count ] ; then
    # handle other arguments
    let original_arg_number=$arg_pos-$short_param_count+1
    local new_arg="${!original_arg_number}"
else
    # break out our one argument into new ones
    local new_arg="-${nodash:$str_pos:1}"
fi
new_args="$new_args \"$new_arg\""
done
# remove the preceding space and set the new arguments
eval set -- "${new_args# }"
fi
if ! $breakout ; then
    key="$nodash"
fi

# By default we expect to shift one argument at a time
shift_count=1
if [ "${#param_pair[@]}" -gt "1" ] ; then
    # This is a param with equals notation
    value="${param_pair[1]}"
else
    # This is either a boolean param and there is no value,
    # or the value is the next command line argument
    # Assume the value is a boolean true, unless the next argument is found to be a
value.
    value=true
    if [[ $# -gt 1 && -n "$2" ]]; then
        local nodash="${2#-}"
        if [ "$nodash" = "$2" ]; then
            # The next argument has NO preceding dash so it is a value
            value="$2"
            shift_count=2
        fi
    fi
fi

# Check that the param being passed is one of the allowed params
if _param_variant "$allowed_params" "$key" ; then
    # --key-name will now become param_key_name
    eval param_${key//-/}_="$value"
else
    printf 'WARNING: Unknown option (ignored): %s\n' "$1" >&2
fi
shift $shift_count
done
}
```

And use it like so:

```
# Assign defaults for parameters
param_help=false
param_path=$(pwd)
param_file=false
param_image=true
param_image_lossy=true
param_image_lossy_quality=85

# Define the params we will allow
allowed_params="h|?|help p|path f|file i|image image-lossy image-lossy-quality"

# Get the params from arguments provided
_get_params $*
```

I don't have enough rep yet to comment or vote his solution up, but [sme's answer](#) worked extremely well for me. The only issue I ran into was that the arguments end up wrapped in single-quotes (so I have an strip them out).

I also added some example usages and HELP text. I'll included my slightly extended version here:

```
#!/bin/bash

# getopt example
# from: https://stackoverflow.com/questions/402377/using-getopts-in-bash-shell-script-to-
get-long-and-short-command-line-options
HELP_TEXT=\
"  USAGE:\n
    Accepts - and -- flags, can specify options that require a value, and can be in any
order. A double-hyphen (--) will stop processing options.\n\n
    Accepts the following forms:\n\n
    getopt-example.sh -a -b -c value-for-c some-arg\n
    getopt-example.sh -c value-for-c -a -b some-arg\n
    getopt-example.sh -abc some-arg\n
    getopt-example.sh --along --blong --clong value-for-c -a -b -c some-arg\n
    getopt-example.sh some-arg --clong value-for-c\n
    getopt-example.sh
"
```

```
aflag=false
bflag=false
cargument=""
```

```
# options may be followed by one colon to indicate they have a required argument
if ! options=$(getopt -o abc:h\? -l along,blong,help,clong: -- "$@")
then
    # something went wrong, getopt will put out an error message for us
    exit 1
fi

set -- $options

while [ $# -gt 0 ]
do
    case $1 in
        -a|--along) aflag=true ;;
        -b|--blong) bflag=true ;;
        # for options with required arguments, an additional shift is required
        -c|--clong) cargument="$2" ; shift;;
        -h|--help|-\\?) echo -e $HELP_TEXT; exit;;
        (-- ) shift; break;;
        (-*) echo "$0: error - unrecognized option $1" 1>&2; exit 1;;
        (*) break;;
    esac
    shift
done

# to remove the single quotes around arguments, pipe the output into:
# | sed -e "s/^\|'$//g" (just leading/trailing) or | sed -e "s/'//g" (all)

echo aflag=${aflag}
echo bflag=${bflag}
echo cargument=${cargument}

while [ $# -gt 0 ]
do
    echo arg=$1
    shift

    if [[ $aflag == true ]]; then
        echo a is true
    fi
done
```

In order to stay cross-platform compatible, and avoid the reliance on external executables, I ported some code from another language.

I find it very easy to use, here is an example:

```
ArgParser::addArg "[h]elp"      false  "This list"
ArgParser::addArg "[q]uiet"     false  "Supress output"
ArgParser::addArg "[s]leep"     1      "Seconds to sleep"
ArgParser::addArg "v"           1      "Verbose mode"

ArgParser::parse "$@"

ArgParser::isset help && ArgParser::showArgs

ArgParser::isset "quiet" \
    && echo "Quiet!" \
    || echo "Noisy!"

local __sleep
ArgParser::tryAndGetArg sleep into __sleep \
    && echo "Sleep for $__sleep seconds" \
    || echo "No value passed for sleep"

# This way is often more convenient, but is a little slower
echo "Sleep set to: $( ArgParser::getArg sleep )"
```

The required BASH is a little longer than it could be, but I wanted to avoid reliance on BASH 4's associative arrays. You can also download this directly from <http://nt4.com/bash/argparser.inc.sh>

```
#!/usr/bin/env bash

# Updates to this script may be found at
# http://nt4.com/bash/argparser.inc.sh

# Example of runtime usage:
# mnc.sh --nc -q Caprica.S0*mkv *.avi *.mp3 --more-options here --host centos8.host.com

# Example of use in script (see bottom)
# Just include this file in yours, or use
# source argparser.inc.sh

unset EXPLODED
declare -a EXPLODED
function explode
{
    local c=$#
    (( c < 2 )) &&
    {
        echo function "$0" is missing parameters
        return 1
    }

    local delimiter="$1"
    local string="$2"
```



```
local limit=${3-99}

local tmp_delim='${\x07}'
local delin=${string//$delimiter/$tmp_delim}
local oldifs="$IFS"

IFS="$tmp_delim"
EXPLODED=($delin)
IFS="$oldifs"
}
```

```
# See: http://fvue.nl/wiki/Bash:\_Passing\_variables\_by\_reference
# Usage: local "$1" && upvar $1 "value(s)"
```

```
upvar() {
    if unset -v "$1"; then          # Unset & validate varname
        if (( $# == 2 )); then
            eval $1="\${2}"        # Return single value
        else
            eval $1=\("${@:2}" )    # Return array
        fi
    fi
}
```

```
function decho
{
    :
}
```

```
function ArgParser::check
{
    __args=${#__argparser__arglist[@]}
    for (( i=0; i<__args; i++ ))
    do
        matched=0
        explode "|" "${__argparser__arglist[$i]}"
        if [ "${#1}" -eq 1 ]
        then
            if [ "${1}" == "${EXPLODED[0]}" ]
            then
                decho "Matched $1 with ${EXPLODED[0]}"
                matched=1

                break
            fi
        else
            if [ "${1}" == "${EXPLODED[1]}" ]
            then
                decho "Matched $1 with ${EXPLODED[1]}"
                matched=1

                break
            fi
        fi
    done
    (( matched == 0 )) && return 2
    # decho "Key $key has default argument of ${EXPLODED[3]}"
    if [ "${EXPLODED[3]}" == "false" ]
    then
        return 0
    else
        return 1
    fi
}
```

```
function ArgParser::set
{
    key=$3
    value="${1:-true}"
    declare -g __argpassed__$key="$value"
}
```

```
function ArgParser::parse
{
    unset __argparser__argv
    __argparser__argv=()
    # echo parsing: "$@"

    while [ -n "$1" ]
    do
        # echo "Processing $1"
        if [ "${1:0:2}" == '--' ]
        then
            key=${1:2}
            value=$2
        elif [ "${1:0:1}" == '-' ]
        then
            key=${1:1}          # Strip off leading -
            value=$2
        else
            decho "Not argument or option: '$1'" >& 2
            __argparser__argv+=( "$1" )
            shift
            continue
        fi
        # parameter=${tmp%*=*}    # Extract name.
        # value=${tmp##*=}        # Extract value.
        decho "Key: '$key', value: '$value'"
        # eval $parameter=$value
    done
}
```

```

    ArgParser::check $key
    el=$?
    # echo "Check returned $el for $key"
    [ $el -eq 2 ] && decho "No match for option '$1'" >&2 # && __argparser__argv+=(
"$1" )
    [ $el -eq 0 ] && decho "Matched option '${EXPLODED[2]}' with no arguments"
>&2 && ArgParser::set true "${EXPLODED[@]}"
    [ $el -eq 1 ] && decho "Matched option '${EXPLODED[2]}' with an argument of
'$2'" >&2 && ArgParser::set "$2" "${EXPLODED[@]}" && shift
    shift
done
}

function ArgParser::isset
{
    declare -p "__argpassed__$1" > /dev/null 2>&1 && return 0
    return 1
}

function ArgParser::getArg
{
    # This one would be a bit silly, since we can only return non-integer arguments
ineffeciently
    varname="__argpassed__$1"
    echo "${!varname}"
}

##
# usage: tryAndGetArg <argname> into <varname>
# returns: 0 on success, 1 on failure
function ArgParser::tryAndGetArg
{
    local __varname="__argpassed__$1"
    local __value="${!__varname}"
    test -z "$__value" && return 1
    local "$3" && upvar $3 "$__value"
    return 0
}

function ArgParser::__construct
{
    unset __argparser__arglist
    # declare -a __argparser__arglist
}

##
# @brief add command line argument
# @param 1 short and/or Long, eg: [s]hort
# @param 2 default value
# @param 3 description
##
function ArgParser::addArg
{
    # check for short arg within long arg
    if [[ "$1" =~ \[(.)\] ]]
    then
        short=${BASH_REMATCH[1]}
        long=${1/\[${short}\]/$short}
    else
        long=$1
    fi
    if [ "${#long}" -eq 1 ]
    then
        short=$long
        long=''
    fi
    decho short: "$short"
    decho long: "$long"
    __argparser__arglist+=("$short|$long|$1|$2|$3")
}

##
# @brief show available command line arguments
##
function ArgParser::showArgs
{
    # declare -p | grep argparser
    printf "Usage: %s [OPTION...]\n\n" "$( basename "${BASH_SOURCE[0]}" )"
    printf "Defaults for the options are specified in brackets.\n\n";

    __args=${#__argparser__arglist[@]}
    for (( i=0; i<__args; i++ ))
    do
        local shortname=
        local fullname=
        local default=
        local description=
        local comma=

        explode "|" "${__argparser__arglist[$i]}"

        shortname="${EXPLODED[0]:+-${EXPLODED[0]}}" # String Substitution Guide:
        fullname="${EXPLODED[1]:+--${EXPLODED[1]}}" #
http://tldp.org/LDP/abs/html/parameter-substitution.html
        test -n "$shortname" \
            && test -n "$fullname" \
            && comma=", "

        default="${EXPLODED[3]}"
        case $default in

```

```
        false )
            default=
            ;;
        "" )
            default=
            ;;
        * )
            default="[$default]"
    esac

    description="${EXPLODED[4]}"

    printf " %2s%1s %-19s %s %s\n" "$shortname" "$comma" "$fullname" "$description"
"$default"
    done
}

function ArgParser::test
{
    # Arguments with a default of 'false' do not take paramaters (note: default
    # values are not applied in this release)

    ArgParser::addArg "[h]elp"      false      "This list"
    ArgParser::addArg "[q]uiet" false      "Supress output"
    ArgParser::addArg "[s]leep" 1          "Seconds to sleep"
    ArgParser::addArg "v"        1          "Verbose mode"

    ArgParser::parse "$@"

    ArgParser::isset help && ArgParser::showArgs

    ArgParser::isset "quiet" \
        && echo "Quiet!" \
        || echo "Noisy!"

    local __sleep
    ArgParser::tryAndGetArg sleep into __sleep \
        && echo "Sleep for $__sleep seconds" \
        || echo "No value passed for sleep"

    # This way is often more convenient, but is a little slower
    echo "Sleep set to: $( ArgParser::getArg sleep )"

    echo "Remaining command line: ${__argparser__argv[@]}"
}

if [ "$( basename "$0" )" == "argparser.inc.sh" ]
then
    ArgParser::test "$@"
fi
```

If all your long options have unique, and matching, first characters as the short options, so for example

```
./slamm --chaos 23 --plenty test -quiet
```

Is the same as

```
./slamm -c 23 -p test -q
```

You can use this *before* getopts to rewrite \$args:

```
# change long options to short options

for arg; do
    [[ "${arg:0:1}" == "-" ]] && delim="" || delim="\\"
    if [ "${arg:0:2}" == "--" ];
    then args="${args} -${arg:2:1}"
    else args="${args} ${delim}${arg}${delim}"
    fi
done

# reset the incoming args
eval set -- $args

# proceed as usual
while getopts ":b:la:h" OPTION; do
    .....
```

Thanks for mtvee for the inspiration ;-)

I don't get the significance of eval here – user.friendly Jun 15 '17 at 22:25

Builtin `getopts` only parse short options (except in ksh93), but you can still add few lines of scripting to make `getopts` handles long options.

Here is a part of code found in <http://www.uxora.com/unix/shell-script/22-handle-long-options-with-getopts>

```
##= set short options ==#
SCRIPT_OPTS=':fbF:B::-h'
##= set Long options associated with short one ==#
```

```
typeset -A ARRAY_OPTS
ARRAY_OPTS=(
  [foo]=f
  [bar]=b
  [foobar]=F
  [barfoo]=B
  [help]=h
  [man]=h
)

#== parse options ==#
while getopt ${SCRIPT_OPTS} OPTION ; do
  #== translate long options to short ==#
  if [[ "x$OPTION" == "x-" ]]; then
    LONG_OPTION=$OPTARG
    LONG_OPTARG=$(echo $LONG_OPTION | grep "=" | cut -d'=' -f2)
    LONG_OPTIND=-1
    [[ "x$LONG_OPTARG" = "x" ]] && LONG_OPTIND=$OPTIND || LONG_OPTION=$(echo $OPTARG
| cut -d'=' -f1)
    [[ $LONG_OPTIND -ne -1 ]] && eval LONG_OPTARG="\${$LONG_OPTIND}"
    OPTION=${ARRAY_OPTS[$LONG_OPTION]}
    [[ "x$OPTION" = "x" ]] && OPTION="?" OPTARG="-${LONG_OPTION}"

    if [[ $( echo "${SCRIPT_OPTS}" | grep -c "${OPTION}:" ) -eq 1 ]]; then
      if [[ "x${LONG_OPTARG}" = "x" ]] || [[ "${LONG_OPTARG}" = -* ]]; then
        OPTION=":" OPTARG="-${LONG_OPTION}"
      else
        OPTARG="${LONG_OPTARG}";
        if [[ $LONG_OPTIND -ne -1 ]]; then
          [[ $OPTIND -le $Optnum ]] && OPTIND=$(( $OPTIND+1 ))
          shift $OPTIND
          OPTIND=1
        fi
      fi
    fi

    fi

  #== options follow by another option instead of argument ==#
  if [[ "x${OPTION}" != "x:" ]] && [[ "x${OPTION}" != "x?" ]] && [[ "${OPTARG}" = -*
]]; then
    OPTARG="$OPTION" OPTION=":"
  fi

  #== manage options ==#
  case "$OPTION" in
    f ) foo=1 bar=0 ;;
    b ) foo=0 bar=1 ;;
    B ) barfoo=${OPTARG} ;;
    F ) foobar=1 && foobar_name=${OPTARG} ;;
    h ) usagefull && exit 0 ;;
    : ) echo "${SCRIPT_NAME}: -$OPTARG: option requires an argument" >&2 && usage >&2
&& exit 99 ;;
    ? ) echo "${SCRIPT_NAME}: -$OPTARG: unknown option" >&2 && usage >&2 && exit 99
;;
    esac
done
shift $(( ${OPTIND} - 1 ))
```

Here is a test:

```
# Short options test
$ ./foobar_any_getopts.sh -bF "Hello world" -B 6 file1 file2
foo=0 bar=1
barfoo=6
foobar=1 foobar_name=Hello world
files=file1 file2

# Long and short options test
$ ./foobar_any_getopts.sh --bar -F Hello --barfoo 6 file1 file2
foo=0 bar=1
barfoo=6
foobar=1 foobar_name=Hello
files=file1 file2
```

Otherwise in recent Korn Shell ksh93, getopt can naturally parse long options and even display a man page alike. (See <http://www.uxora.com/unix/shell-script/20-getopts-with-man-page-and-long-options>)

Th built-in OS X (BSD) getopt does not support long options, but the GNU version does: brew install gnu-getopt . Then, something similar to: cp /usr/local/Cellar/gnu-getopt/1.1.6/bin/getopt /usr/local/bin/gnu-getopt .

EasyOptions handles short and long options:

```
## Options:
## --verbose, -v Verbose mode
## --logfile=NAME Log filename

source easyoptions || exit

if test -n "${verbose}"; then
  echo "log file: ${logfile}"
  echo "arguments: ${arguments[@]}"
fi
```

getopts "could be used" for parsing long options as long as you don't expect them to have arguments...

Here's how to:

```
$ cat > longopt
while getopts 'e:--' OPT; do
  case $OPT in
    e) echo echo: $OPTARG;;
    -) #Long option
      case $OPTARG in
        long-option) echo long option;;
        *) echo long option: $OPTARG;;
      esac;;
    esac
  done

$ bash longopt -e asd --long-option --long1 --long2 -e test
echo: asd
long option
long option: long1
long option: long2
echo: test
```

If you try to use OPTIND for getting a parameter for the long option, getopt will treat it as the first no optional positional parameter and will stop parsing any other parameters. In such a case you'll be better off handling it manually with a simple case statement.

This will "always" work:

```
$ cat >longopt2
while (($#)); do
  OPT=$1
  shift
  case $OPT in
    --*) case ${OPT:2} in
      long1) echo long1 option;;
      complex) echo complex with argument $1; shift;;
    esac;;

    -*) case ${OPT:1} in
      a) echo short option a;;
      b) echo short option b with parameter $1; shift;;
    esac;;
  esac
done

$ bash longopt2 --complex abc -a --long -b test
complex with argument abc
short option a
short option b with parameter test
```

Albeit is not as flexible as getopt and you have to do much of the error checking code yourself within the case instances...

But it is an option.

hm.

not really satisfied with the pure bash options. why not use perl to get what you want. Directly parse the \$* array, and auto-name your options.

simple helper script:

```
#!/usr/bin/perl
use Getopt::Long;

my $optstring = shift;

my @opts = split(m#,#, $optstring);

my %opt;
GetOptions(\%opt, @opts);

print "set -- " . join(' ', map("'$_'", @ARGV)) . " ";
my $xx;

my $key;
foreach $key (keys(%opt))
{
  print "export $key='${opt}{$key}'; ";
}


```

then you can use in your script as a one liner, for example:

```
#!/bin/bash

eval `getopts.pl reuse:s,long_opt:s,hello $*`;

echo "HELLO: $hello"
echo "LONG_OPT: $long_opt"
echo "REUSE: $reuse"
```



```
echo $*

/tmp/script.sh hello --reuse me --long_opt whatever_you_want_except_spaces --hello 1 2 3

HELLO: 1 LONG_OPT: whatever_you_want_except_spaces REUSE: me

1 2 3
```

Only caveat here is spaces don't work. But it avoids bash's rather complicated looping syntax, works with long args, auto-names them as variables and automatically resizes \$*, so will work 99% of the time.

This is flawed. You need to use "\$@" rather than \$* to correctly preserve whitespace and other shell metacharacter in the list of arguments. (I have not examined the code any further.) – tripleee May 19 '15 at 13:34

protected by codeforester Aug 1 at 5:13

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?