Avoid BSDM*: Obey the command. Line, that is.

Steven Lembark
Workhorse Computing
lembark@wrkhors.com

**BSD Masochism*

Following commands

Lots of ways to handle the commnd line.

Most of them are terrible.

getopt(1) in all its varieties is the sanest.

Standardize the comand line structure.

But it only gets you so far.

Following commands

This is about BASH.

We’ll use:

- getopt
- while
- case
- associatiave arrays
- functions

Tie it all up nicely, so to speak.

Where to find the commands.

BASH stores arguments in an array.

‘Options’ have ‘-’ or ‘–’ and may have arg’s.

Then there are args.

In *whatever* order they were typed.

foo --bletch=blort --bim /my/path -d bam -v1 -x;

Where to find the commands.

BASH stores arguments in an array.

‘Options’ have ‘-’ or ‘–’ and may have arg’s.

Then there are args.

In *whatever* order they were typed.

foo --bletch=blort --bim /my/path -d bam -v1 -x;

Where to find the commands.

BASH stores arguments in an array.

‘Options’ have ‘-’ or ‘–’ and may have arg’s.

Then there are args.

In *whatever* order they were typed.

foo --bletch=blort --bim /my/path -d bam -v1 -x;
foo --bletch=blort --bim /my/path -d bam -v1 -x;

Where to find the commands.

BASH stores arguments in an array.

‘Options’ have ‘-’ or ‘–’ and may have arg’s.

Then there are args.

In *whatever* order they were typed.

foo --bletch=blort --bim /my/path -d bam -v1 -x;
foo --bletch=blort --bim /my/path -d bam -v1 -x;
foo --bletch=blort --bim /my/path -d bam -v1 -x;

getopt(3) to the rescue!!!

C function.

List of possible switches.

Switches may have arguments.

Switches extracted before returning arguments.

Metadata describes valid switches, their args.

Processes all of the switches, leaving program args.

getopt(3) to the rescue!!!

C function.

getopt_long(3) allowed for “--foo” and “--foo=bar”.

Named switches!!!

Every language since has been getopt_long[ish].

Where to find the commands.

BASH stores arguments in an array...

In *whatever* order they were typed.

Two ways to access them: $* and $@.

Behave differently when quoted:

- ”$*” works seprated by $IFS.
- ”$@” separate words.

$* for printing, $@ is for iterating.

Where to find the commands.

The program and sub’s all use $* and $@.

Localized by context.

Result: You cannot access program arg’s from a sub.

Need to pass them as “$@”.

Guts of getopt(1)

BASH added a getopt command line utility.

It’s arguments define the switches.

Two types:

Short are dash and letter, may be combined.

-v 1 -d bam -x

-v1 -dbam x

-vdx 1 bam

Fun eh?

Guts of getopt(1)

BASH added a getopt command line utility.

It’s arguments define the switches.

Two types:

Long arguments are double-dash and optional ‘=’

--bletch=blort

--blecth blort

No stacking, argument always next item.

Guts of getopt(1)

BASH added a getopt command line utility.

In short: USE LONG ARGUMENTS!!!

Only place short’s make sense:

- Binary (on/off) behavior without args.
- “-d” for debug, “-v” for verbose.

BASH getopt(1): First pass.

Say you don’t understand arrays or functions.

Simplest case: Stuff it all on one line.

Options with argumens get a ‘:’.

There are ways of typing the arguments also.

```
eval set -- "$( getopt -o’d:v:x’ -l’--blort:,--bim’ -- "$@" )";
```

BASH getopt(1): First pass.

Say you don’t understand arrays or functions.

Simplest case: Stuff it all on one line.

“$@” gives you separate words.

The ‘--’ separates getopt’s args from yours.

```
eval set -- "$( getopt -o’d:v:x’ -l’--blort:,--bim’ -- "$@" )";
```

BASH getopt(1): First pass.

Say you don’t understand arrays or functions.

Simplest case: Stuff it all on one line.

$(…) executes a command, returns the result.

“set --” re-sets the program arguments.

```
eval set -- "$( getopt -o’d:v:x’ -l’--blort:,--bim’ -- "$@" )";
```

BASH getopt(1): First pass.

Say you don’t understand arrays or functions.

Simplest case: Stuff it all on one line.

eval post-processes the mess returned by getopt.

```
eval set -- "$( getopt -o’d:v:x’ -l’--blort:,--bim’ -- "$@" )";
```

BASH getopt(1): First pass.

foo --bletch=blort --bim /my/path -d bam -v1 -x;

becomes:

“--bletch blort --bim -d bam -v1 -x -- /my/path”

```
eval set -- "$( getopt -o’d:v:x’ -l’--blort:,--bim’ -- "$@" )";
```

BASH getopt(1): First pass.

foo --bletch=blort --bim /my/path -d bam -v1 -x;

becomes:

“--bletch blort --bim -d bam -v1 -x -- /my/path”

The -- is a Very Good Thing(tm).

Separates options from program arguments.

```
eval set -- "$( getopt -o’d:v:x’ -l’--blort:,--bim’ -- "$@" )";
```

BASH getopt(1): First pass.

foo --bletch=blort --bim /my/path -d bam -v1 -x;

becomes:

“--bletch blort --bim -d bam -v1 -x -- /my/path”

A simple loop iterates options.

Terminates on the first ‘--’.

```
eval set -- "$( getopt -o’d:v:x’ -l’--blort:,--bim’ -- "$@" )";
```

BASH getopt(1): Second pass.

A bit more readable.

At least descriptive...

```
short=’d:v:x’;
 long=’blort:,bim’;

argv="$( getopt -o”$short” -l”$long” -- "$@" )";
eval set -- “$argv”;
```

BASH getopt(1): Second pass.

A bit more readable.

Places to hang error messages.

```
short=’d:v:x’;
 long=’--blort:,--bim’;

argv="$( getopt -o”$short” -l”$long” -- "$@" )";
[[ “$argv” =~ unrecognized ]] && usage “Bogus options: $argv”;

eval set -- “$argv”;
```

BASH getopt(1): Processing argv

Now what?

Simplest way is iterating “$@”.

where and a case.

Keep going until ‘--’.

```
eval set -- “$argv”;

--bletch blort --bim -d bam -v1 -x -- /my/path
```

BASH getopt(1): Processing argv

```
eval set -- “$argv”;
while :;
do
   case $1           # $1, $2 are elements of $@/$*.
   in
      --)   shift;   # discard the --
            break;   # program args left on $@
      ;;
      --bletch)      # this has an argument
            bletch=$2;
            shift;
      ;;
   ...
   esac
   shift;           # discard $1.
done
```

BASH getopt(1): Processing argv

Catch: This fills your code with global variables.

$bletch, $verobse, $debug...

```
bletch=$2;
```

BASH getopt(1): Third pass

Fix: Push getopt into a subroutine.

```
command-line()
{
   # $@/$@ are always local to the context.

   typeset -r short=’d:v:x’;
   typeset -r  long=’blort:,bim’;

   local argv="$( getopt -o”$short” -l”$long” -- "$@" )";
   eval set -- “$argv”;
```

BASH getopt(1): Third pass

Fix: Push getopt into a subroutine.

typeset localizes the variables.

```
command-line()
{
   # $@/$@ are always local to the context.

   typeset -r short='d:v:x';
   typeset -r  long='blort:,bim';

   local argv="$( getopt -o"$short" -l"$long" -- "$@" )";
   eval set -- "$argv";
```

BASH getopt(1): Third pass

Fix: Push getopt into a subroutine.

typeset localizes the variables.

-r marks them as read-only.

```
command-line()
{
   # $@/$@ are always local to the context.

   typeset -r short='d:v:x';
   typeset -r  long='blort:,bim';

   local argv="$( getopt -o"$short" -l"$long" -- "$@" )";
   eval set -- "$argv";
```

BASH getopt(1): Third pass

Fix: Push getopt into a subroutine.

$*/$@ are always local to their context.

```
command-line()
{
   # $@/$@ are always local to the context.

   typeset -r short=’d:v:x’;
   typeset -r  long=’blort:,bim’;

   local argv="$( getopt -o”$short” -l”$long” -- "$@" )";
   eval set -- “$argv”;
```

BASH getopt(1): Third pass

How do we handle the options?

TMTOWTDI!!!

```
#!/bin/bash

command-line()
{
    ...

}
```

BASH getopt(1): Third pass

Pre-declare the results.

Simpler processing with lower-case values.

```
#!/bin/bash
typeset -l bletch;   # pre-declare globals.

command-line()
{
      $bletch = $2;  # “true”, not True TRue TRUE truE
}
```

BASH getopt(1): Third pass

How do we handle the options?

Pass them back as a list.

```
#!/bin/bash

command-line()
{
    ...
    ( $bletch, $bim, $verbose, $debug )
}

typeset -a argv=( $(command-line “$@”) );
```

BASH getopt(1): Third pass

How do we handle the options?

Downside: You have to remember the order.

```
#!/bin/bash

command-line()
{
   ...
   ( $bletch, $bim, $verbose, $debug )
}

typeset -a argv=( $(command-line “$@”) );
```

BASH getopt(1): Third pass

How do we handle the options?

Better way: Associative array

```
#!/bin/bash

command-line()
{
   --blort)
         $blort=$2;
         shift;
   ;;
}
```

BASH getopt(1): Third pass

How do we handle the options?

Better way: Associative array

```
#!/bin/bash

command-line()
{
    --blort)
          ${argv[blort]}=$2;
          shift;
    ;;
}
```

BASH getopt(1): Third pass

How do we handle the options?

${foobar[1]} indexed array

${foobar[X]} associative array (map/hash/dictionary)

```
--blort)
   ${argv[blort]}=$2;
   shift;
;;
-v)
   ${argv[verbose]}=1;
;;
```

BASH getopt(1): Third pass

How do we handle the options?

Simply declare a global $argv.

```
typeset -A argv=();     # predeclare it, look organized!

command-line()
{
   ${argv[blort]}=$2;

   ${argv[verbose]}=1;
}
```

BASH getopt(1): Third pass

How do we handle the options?

Anything that isn't localized is global.

```
~~typeset -A argv=();~~

command-line()
{
   ${argv[blort]}=$2;   # assigning to ${argv[X]} creates argv.

   ${argv[verbose]}=1;
}

if [[ -n ${argv[verbose]} ]]
```

BASH getopt(1): Third pass

Er... What happened to the program args?

Pass them back as a list!

```
command-line()
{
   ${argv[blort]}=$2;   # assigning to ${argv[X]} creates argv.

   ${argv[verbose]}=1;

   "$@"                 # hand back what's left after shift.
}
# read-only variable cmd_arg has what's left after the --
typeset -r -a cmd_arg=( $( command-line "$@" ) );
```

BASH getopt(1): Fourth pass

Better error messages with -n and a program name.

base0 will be used with the getopt error messages.

```
#!/bin/bash
typeset -r base0=$( basename $0 );
typeset -r  dir0=$( dirname  $0 );

local argv="$( getopt -o"$short" -l"$long" -n"$base0" -- "$@" )";
```

Re-usable command line handler for BASH

One sub to rule them all.

Or at least command them...

```
typeset -r -a cmd_args=$( command-line “$@” );

[[ -n $cmd_args ]] || fatal “Bogus $base0: no arguments”;

validate-source-file ${cmd_args[0]};

[[ -n ${argv[verbose]} ]] && echo “Verbosely...”
```

Bedside reading

```
man 1 bash;      # arrays, typeset.
man 1 getopt;    # BASH getopt
man 3 getopt;    # where it all came from
```