Handling positional parameters

Intro

The day will come when you want to give arguments to your scripts. These arguments are known as **positional parameters**. Some relevant special parameters are described below:

Parameter(s)	Description
`$0`	the first positional parameter, equivalent to `argv[0]` in C, see **the first argument**
`$FUNCNAME`	the function name (**attention**: inside a function, `$0` is still the `$0` of the shell, **not** the function name)
`$1 … $9`	the argument list elements from 1 to 9
`${10} … ${N}`	the argument list elements beyond 9 (note the parameter expansion syntax!)
`$*`	all positional parameters except `$0`, see **mass usage**
`$@`	all positional parameters except `$0`, see **mass usage**
`$#`	the number of arguments, not counting `$0`

These positional parameters reflect exactly what was given to the script when it was called.

Option-switch parsing (e.g. `-h` for displaying help) is not performed at this point.

See also the dictionary entry for "parameter".

The first argument

The very first argument you can access is referenced as `$0`. It is usually set to the script's name exactly as called, and it's set on shell initialization:

Testscript - it just echos `$0`:

```
#!/bin/bash
echo "$0"
```

You see, `$0` is always set to the name the script is called with (`>` is the prompt…):

```
> ./testscript
./testscript
```

```
> /usr/bin/testscript
/usr/bin/testscript
```

However, this isn't true for login shells:

```
> echo "$0"
-bash
```

In other terms, `$0` is not a positional parameter, it's a special parameter independent from the positional parameter list. It can be set to anything. In the **ideal** case it's the pathname of the script, but since this gets set on invocation, the invoking program can easily influence it (the `login` program does that for login shells, by prefixing a dash, for example).

Inside a function, `$0` still behaves as described above. To get the function name, use `$FUNCNAME`.

Shifting

The builtin command `shift` is used to change the positional parameter values:

- `$1` will be discarded
- `$2` will become `$1`
- `$3` will become `$2`
- ...
- in general: `$N` will become `$N-1`

The command can take a number as argument: Number of positions to shift. e.g. `shift 4` shifts `$5` to `$1`.

Using them

Enough theory, you want to access your script-arguments. Well, here we go.

One by one

One way is to access specific parameters:

```
#!/bin/bash
echo "Total number of arguments: $#"
echo "Argument 1: $1"
echo "Argument 2: $2"
echo "Argument 3: $3"
echo "Argument 4: $4"
echo "Argument 5: $5"
```

While useful in another situation, this way is lacks flexibility. The maximum number of arguments is a fixedvalue - which is a bad idea if you write a script that takes many filenames as arguments.

⇒ forget that one

Loops

There are several ways to loop through the positional parameters.

You can code a C-style for-loop using `$#` as the end value. On every iteration, the `shift` -command is used to shift the argument list:

```
numargs=$#
for ((i=1 ; i <= numargs ; i++))
do
    echo "$1"
    shift
done
```

Not very stylish, but usable. The `numargs` variable is used to store the initial value of `$#` because the shift command will change it as the script runs.

Another way to iterate one argument at a time is the `for` loop without a given wordlist. The loop uses the positional parameters as a wordlist:

```
for arg
do
    echo "$arg"
done
```

Advantage: The positional parameters will be preserved

The next method is similar to the first example (the `for` loop), but it doesn't test for reaching `$#` . It shifts and checks if `$1` still expands to something, using the test command:

```
while [ "$1" ]
do
    echo "$1"
    shift
done
```

Looks nice, but has the disadvantage of stopping when `$1` is empty (null-string). Let's modify it to run as long as `$1` is defined (but may be null), using parameter expansion for an alternate value:

```
while [ "${1+defined}" ]; do
  echo "$1"
  shift
done
```

Getopts

There is a small tutorial dedicated to ''getopts'' (*under construction*).

Mass usage

All Positional Parameters

Sometimes it's necessary to just "relay" or "pass" given arguments to another program. It's very inefficient to do that in one of these loops, as you will destroy integrity, most likely (spaces!).

The shell developers created `$*` and `$@` for this purpose.

As overview:

Syntax	Effective result
`$*`	`$1 $2 $3 … ${N}`
`$@`	`$1 $2 $3 … ${N}`
`"$*"`	`"$1c$2c$3c…c${N}"`
`"$@"`	`"$1" "$2" "$3" … "${N}"`

Without being quoted (double quotes), both have the same effect: All positional parameters from `$1` to the last one used are expanded without any special handling.

When the `$*` special parameter is double quoted, it expands to the equivalent of: `"$1c$2c$3c$4c……..$N"` , where 'c' is the first character of `IFS` .

But when the `$@` special parameter is used inside double quotes, it expands to the equivanent of…

`"$1" "$2" "$3" "$4" ….. "$N"`

…which **reflects all positional parameters as they were set initially** and passed to the script or function. If you want to re-use your positional parameters to **call another program** (for example in a wrapper-script), then this is the choice for you, use double quoted `"$@"` .

Well, let's just say: **You almost always want a quoted `"$@"` !**

Range Of Positional Parameters

Another way to mass expand the positional parameters is similar to what is possible for a range of characters using substring expansion on normal parameters and the mass expansion range of arrays.

`${@:START:COUNT}`

`${*:START:COUNT}`

`"${@:START:COUNT}"`

`"${*:START:COUNT}"`

The rules for using `@` or `*` and quoting are the same as above. This will expand `COUNT` number of positional parameters beginning at `START` . `COUNT` can be omitted (`${@:START}`), in which case, all positional parameters beginning at `START` are expanded.

If `START` is negative, the positional parameters are numbered in reverse starting with the last one.

`COUNT` may not be negative, i.e. the element count may not be decremented.

Example: START at the last positional parameter:

```
echo "${@: -1}"
```

Attention: As of Bash 4, a `START` of `0` includes the special parameter `$0` , i.e. the shell name or whatever $0 is set to, when the positional parameters are in use. A `START` of `1` begins at `$1` . In Bash 3 and older, both `0` and `1` began at `$1` .

Setting Positional Parameters

Setting positional parameters with command line arguments, is not the only way to set them. The builtin command, set may be used to "artificially" change the positional parameters from inside the script or function:

```
set "This is" my new "set of" positional parameters

# RESULTS IN
# $1: This is
# $2: my
# $3: new
# $4: set of
# $5: positional
# $6: parameters
```

It's wise to signal "end of options" when setting positional parameters this way. If not, the dashes might be interpreted as an option switch by `set` itself:

```
# both ways work, but behave differently. See the article about the s
et command!
set -- ...
set - ...
```

Alternately this will also preserve any verbose (-v) or tracing (-x) flags, which may otherwise be reset by `set`

```
set -$- ...
```

Fix Me! continue

Production examples

Using a while loop

To make your program accept options as standard command syntax:

`COMMAND [options] <params>` # Like 'cat -A file.txt'

See simple option parsing code below. It's not that flexible. It doesn't auto-interpret combined options (-fu USER) but it works and is a good rudimentary way to parse your arguments.

```
#!/bin/sh
# Keeping options in alphabetical order makes it easy to add more.

while :
do
    case "$1" in
      -f | --file)
          file="$2"   # You may want to check validity of $2
          shift 2
          ;;
      -h | --help)
          display_help  # Call your function
          # no shifting needed here, we're done.
          exit 0
          ;;
      -u | --user)
          username="$2" # You may want to check validity of $2
          shift 2
          ;;
      -v | --verbose)
          #  It's better to assign a string, than a number like "verb
ose=1"
          #  because if you're debugging the script with "bash -x" co
de like this:
          #
          #    if [ "$verbose" ] ...
          #
          #  You will see:
          #
          #    if [ "verbose" ] ...
          #
          #  Instead of cryptic
          #
          #    if [ "1" ] ...
          #
          verbose="verbose"
          shift
          ;;
      --) # End of all options
          shift
          break;
      -*)
          echo "Error: Unknown option: $1" >&2
          exit 1
          ;;
      *)  # No more options
          break
          ;;
    esac
done

# End of file
```

Filter unwanted options with a wrapper script

This simple wrapper enables filtering unwanted options (here: `-a` and `–all` for `ls`) out of the command line. It reads the positional parameters and builds a filtered array consisting of them, then calls `ls` with the new option set. It also respects the `–` as "end of options" for `ls` and doesn't change anything after it:

```
#!/bin/bash

# simple ls(1) wrapper that doesn't allow the -a option

options=()  # the buffer array for the parameters
eoo=0       # end of options reached

while [[ $1 ]]
do
    if ! ((eoo)); then
        case "$1" in
          -a)
              shift
              ;;
          --all)
              shift
              ;;
          -[^-]*a*|-a?*)
              options+=("${1//a}")
              shift
              ;;
          --)
              eoo=1
              options+=("$1")
              shift
              ;;
          *)
              options+=("$1")
              shift
              ;;
        esac
    else
        options+=("$1")

        # Another (worse) way of doing the same thing:
        # options=("${options[@]}" "$1")
        shift
    fi
done

/bin/ls "${options[@]}"
```

Using getopts

There is a small tutorial dedicated to ''getopts'' (*under construction*).

See also

- Internal: Small getopts tutorial
- Internal: The while-loop
- Internal: The C-style for-loop
- Internal: Arrays (for equivalent syntax for mass-expansion)
- Internal: Substring expansion on a parameter (for equivalent syntax for mass-expansion)
- Dictionary, internal: Parameter

Discussion

skmdu, 2010/04/14 12:20 (), 2010/04/14 15:13 ()

> The shell-developers invented $* and $@ for this purpose.

> Without being quoted (double-quoted), both have the same effect: All positional parameters from $1 to the last used one >are expanded, separated by the first character of IFS (represented by "c" here, but usually a space): $1c$2c$3c$4c.....$N

Without double quotes, $* and $@ are expanding the positional parameters separated by only space, not by IFS.

```
#!/bin/bash

export IFS='-'

echo -e $*
echo -e $@
```

```
$./test "This is" 2 3
This is 2 3
This is 2 3
```

(Edited: Inserted code tags)

Jan Schampera, 2010/04/14 15:12 ()

Thank you very much for this finding. I know how `$*` works, thus I can't understand why I described it that wrong. I guess it was in some late night session.

Thanks again.

gdh, 2011/02/18 15:11 ()

#!/bin/bash

OLDIFS="$IFS" IFS='-' #export IFS='-'

#echo -e $* #echo -e $@ #should be echo -e "$*" echo -e "$@" IFS="$OLDIFS"

gdh, 2011/02/18 15:14 ()

#should be echo -e "$*"

Dave Carlton (http://polymicrosystems.com), 2010/05/18 13:23 ()

I would suggext using a different prompt as the $ is confusing to newbies. Otherwise, an excellent treatise on use of positional parameters.

Jan Schampera, 2010/05/24 08:48 ()

Thanks for the suggestion, I use "> " here now, and I'll change it in whatever text I edit in future (whole wiki). Let's see if "> " is okay.

herb, 2012/04/20 08:32 ()

Here's yet another non-getopts way.

http://bsdpants.blogspot.de/2007/02/option-ize-your-shell-scripts.html (http://bsdpants.blogspot.de/2007/02/option-ize-your-shell-scripts.html)

aborrero, 2012/07/16 12:48 (), 2012/08/12 07:06 ()

Hi there!

What if I use "$@" in subsequent function calls, but arguments are strings?

I mean, having:

```
#!/bin/bash
echo "$@"
echo n: $#
```

If you use it

```
mypc$ script arg1 arg2 "asd asd" arg4
arg1 arg2 asd asd arg4
n: 4
```

But having

```
#!/bin/bash
myfunc()
{
  echo "$@"
  echo n: $#
}
ech "$@"
echo n: $#
myfunc "$@"
```

you get:

```
mypc$ myscrpt arg1 arg2 "asd asd" arg4
arg1 arg2 asd asd arg4
4
arg1 arg2 asd asd arg4
5
```

As you can see, there is no way to make know the function that a parameter is a string and not a space separated list of arguments.

Any idea of how to solve it? I've test calling functions and doing expansion in almost all ways with no results.

Jan Schampera, 2012/08/12 07:11 ()

I don't know why it fails for you. It should work if you use `"$@"` , of course.

See the exmaple I used your second script with:

```
$ ./args1 a b c "d e" f
a b c d e f
n: 5
a b c d e f
n: 5
```

Jacek Puchta, 2015/06/10 08:00 ()

Thanks a lot for this tutorial. Especially the first example is very helpful.

Bash Hackers Wiki