32 – Positional Parameters

One feature that has been missing from our programs so far is the ability to accept and process command line options and arguments. In this chapter, we will examine the shell features that allow our programs to get access to the contents of the command line.

Accessing the Command Line

The shell provides a set of variables called *positional parameters* that contain the individual words on the command line. The variables are named `0` through `9`. They can be demonstrated this way:

```
#!/bin/bash

# posit-param: script to view command line parameters

echo "
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

This is a simple script that displays the values of the variables `$0-$9`. When executed with no command line arguments, the result is this:

```
[me@linuxbox ~]$ posit-param
```

```
$0 = /home/me/bin/posit-param
$1 =
$2 =
$3 =
$4 =
$5 =
$6 =
$7 =
$8 =
$9 =
```

Even when no arguments are provided, `$0` will always contain the first item appearing on the command line, which is the pathname of the program being executed. When arguments are provided, we see these results:

```
[me@linuxbox ~]$ posit-param a b c d

$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

Note: You can actually access more than nine parameters using parameter expansion. To specify a number greater than nine, surround the number in braces as in `${10}`, `${55}`, `${211}`, and so on.

Determining the Number of Arguments

The shell also provides a variable, `$#`, that contains the number of arguments on the command line:

```
#!/bin/bash

# posit-param: script to view command line parameters

echo "
Number of arguments: $#
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

This is the result:

```
[me@linuxbox ~]$ posit-param a b c d

Number of arguments: 4
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

`shift` – Getting Access to Many Arguments

But what happens when we give the program a large number of arguments such as the following?

```
[me@linuxbox ~]$ posit-param *
```

```
Number of arguments: 82
$0 = /home/me/bin/posit-param
$1 = addresses.ldif
$2 = bin
$3 = bookmarks.html
$4 = debian-500-i386-netinst.iso
$5 = debian-500-i386-netinst.jigdo
$6 = debian-500-i386-netinst.template
$7 = debian-cd_info.tar.gz
$8 = Desktop
$9 = dirlist-bin.txt
```

On this example system, the wildcard `*` expands into 82 arguments. How can we process that many? The shell provides a method, albeit a clumsy one, to do this. The `shift` command causes all the parameters to "move down one" each time it is executed. In fact, by using `shift`, it is possible to get by with only one parameter (in addition to `$0`, which never changes).

```
#!/bin/bash

# posit-param2: script to display all arguments

count=1

while [[ $# -gt 0 ]]; do
    echo "Argument $count = $1"
    count=$((count + 1))
    shift
done
```

Each time `shift` is executed, the value of `$2` is moved to `$1`, the value of `$3` is moved to `$2` and so on. The value of `$#` is also reduced by one.

In the `posit-param2` program, we create a loop that evaluates the number of arguments remaining and continues as long as there is at least one. We display the current argument, increment the variable `count` with each iteration of the loop to provide a running count of the number of arguments processed, and, finally, execute a `shift` to load `$1` with the next argument. Here is the program at work:

```
[me@linuxbox ~]$ posit-param2 a b c d
Argument 1 = a
Argument 2 = b
Argument 3 = c
Argument 4 = d
```

Simple Applications

Even without `shift`, it's possible to write useful applications using positional parameters. By way of example, here is a simple file information program:

```
#!/bin/bash

# file-info: simple file information program

PROGNAME="$(basename "$0")"

if [[ -e "$1" ]]; then
    echo -e "\nFile Type:"
    file "$1"
    echo -e "\nFile Status:"
    stat "$1"
else
    echo "$PROGNAME: usage: $PROGNAME file" >&2
    exit 1
fi
```

This program displays the file type (determined by the `file` command) and the file status (from the `stat` command) of a specified file. One interesting feature of this program is the `PROGNAME` variable. It is given the value that results from the `basename "$0"` command. The `basename` command removes the leading portion of a pathname, leaving only the base name of a file. In our example, `basename` removes the leading portion of the pathname contained in the `$0` parameter, the full pathname of our example program. This value is useful when constructing messages such as the usage message at the end of the program. By coding it this way, the script can be renamed, and the message automatically adjusts to contain the name of the program.

Using Positional Parameters with Shell Functions

Just as positional parameters are used to pass arguments to shell scripts, they can also be used to pass arguments to shell functions. To demonstrate, we will convert the file_info script into a shell function.

```
file_info () {

    # file_info: function to display file information

    if [[ -e "$1" ]]; then
         echo -e "\nFile Type:"
         file "$1"
         echo -e "\nFile Status:"
         stat "$1"
    else
         echo "$FUNCNAME: usage: $FUNCNAME file" >&2
         return 1
    fi
}
```

Now, if a script that incorporates the file_info shell function calls the function with a filename argument, the argument will be passed to the function.

With this capability, we can write many useful shell functions that not only can be used in scripts, but also can be used within our .bashrc files.

Notice that the PROGNAME variable was changed to the shell variable FUNCNAME. The shell automatically updates this variable to keep track of the currently executed shell function. Note that $0 always contains the full pathname of the first item on the command line (i.e., the name of the program) and does not contain the name of the shell function as we might expect.

Handling Positional Parameters e*n Masse*

It is sometimes useful to manage all the positional parameters as a group. For example, we might want to write a "wrapper" around another program. This means we create a script or shell function that simplifies the invocation of another program. The wrapper, in this case, supplies a list of arcane command line options and then passes a list of arguments to the lower-level program.

The shell provides two special parameters for this purpose. They both expand into the complete list of positional parameters but differ in rather subtle ways. They are described in Table 32-1.

*Table 32-1: The * and @ Special Parameters*

Parameter	Description
`$*`	Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands into a double-quoted string containing all of the positional parameters, each separated by the first character of the IFS shell variable (by default a space character).
`$@`	Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands each positional parameter into a separate word as if it was surrounded by double quotes.

Here is a script that shows these special parameters in action:

```
#!/bin/bash

# posit-params3: script to demonstrate $* and $@

print_params () {
    echo "\$1 = $1"
    echo "\$2 = $2"
    echo "\$3 = $3"
    echo "\$4 = $4"
}

pass_params () {
    echo -e "\n" '$* :';   print_params $*
    echo -e "\n" '"$*" :'; print_params "$*"
    echo -e "\n" '$@ :';   print_params $@
    echo -e "\n" '"$@" :'; print_params "$@"
}

pass_params "word" "words with spaces"
```

In this rather convoluted program, we create two arguments: `word` and `words with spaces`, and pass them to the `pass_params` function. That function, in turn, passes them on to the `print_params` function, using each of the four methods available with the special parameters `$*` and `$@`. When executed, the script reveals the differences.

```
[me@linuxbox ~]$ posit-param3

 $* :
$1 = word
$2 = words
$3 = with
$4 = spaces

 "$*" :
$1 = word words with spaces
$2 =
$3 =
$4 =

 $@ :
$1 = word
$2 = words
$3 = with
$4 = spaces

 "$@" :
$1 = word
$2 = words with spaces
$3 =
$4 =
```

With our arguments, both `$*` and `$@` produce a four-word result.

```
word words with spaces
```

`"$*"` produces a one-word result:

```
"word words with spaces"
```

`"$@"` produces a two-word result:

```
"word" "words with spaces"
```

This matches our actual intent. The lesson to take from this is that even though the shell provides four different ways of getting the list of positional parameters, `"$@"` is by far the most useful for most situations because it preserves the integrity of each positional parameter. To ensure safety, it should always be used, unless we have a compelling reason not to use it.

A More Complete Application

After a long hiatus, we are going to resume work on our `sys_info_page` program, last seen in Chapter 27. Our next addition will add several command line options to the program as follows:

- **Output file.** We will add an option to specify a name for a file to contain the program's output. It will be specified as either `-f` *`file`* or `--file` *`file`*.
- **Interactive mode.** This option will prompt the user for an output filename and will determine whether the specified file already exists. If it does, the user will be prompted before the existing file is overwritten. This option will be specified by either `-i` or `--interactive`.
- **Help.** Either `-h` or `--help` may be specified to cause the program to output an informative usage message.

Here is the code needed to implement the command line processing:

```
usage () {
    echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
    return
}

# process command line options

interactive=
filename=

while [[ -n "$1" ]]; do
    case "$1" in
         -f | --file)        shift
                             filename="$1"
                             ;;
         -i | --interactive) interactive=1
                             ;;
         -h | --help)        usage
                             exit
                             ;;
         *)                  usage >&2
                             exit 1
                             ;;
    esac
    shift
done
```

First, we add a shell function called `usage` to display a message when the help option is invoked or an unknown option is attempted.

Next, we begin the processing loop. This loop continues while the positional parameter `$1` is not empty. At the end of the loop, we have a `shift` command to advance the positional parameters to ensure that the loop will eventually terminate.

Within the loop, we have a `case` statement that examines the current positional parameter to see whether it matches any of the supported choices. If a supported parameter is found, it is acted upon. If an unknown choice is found the usage message is displayed and the script terminates with an error.

The `-f` parameter is handled in an interesting way. When detected, it causes an additional `shift` to occur, which advances the positional parameter `$1` to the filename argument supplied to the `-f` option.

We next add the code to implement the interactive mode.

```
# interactive mode

if [[ -n "$interactive" ]]; then
    while true; do
         read -p "Enter name of output file: " filename
         if [[ -e "$filename" ]]; then
              read -p "'$filename' exists. Overwrite? [y/n/q] > "
              case "$REPLY" in
                   Y|y) break
                        ;;
                   Q|q) echo "Program terminated."
                        exit
                        ;;
                   *)   continue
                        ;;
              esac
         elif [[ -z "$filename" ]]; then
              continue
         else
              break
         fi
    done
fi
```

If the `interactive` variable is not empty, an endless loop is started, which contains the filename prompt and subsequent existing file-handling code. If the desired output file already exists, the user is prompted to overwrite, choose another filename, or quit the program. If the user chooses to overwrite an existing file, a `break` is executed to terminate the loop. Notice how the `case` statement detects only whether the user chooses to overwrite or quit. Any other choice causes the loop to continue and prompts the user again.

To implement the output filename feature, we must first convert the existing page-writing code into a shell function, for reasons that will become clear in a moment.

```
write_html_page () {
    cat <<- _EOF_
    <html>
         <head>
              <title>$TITLE</title>
         </head>
         <body>
              <h1>$TITLE</h1>
              <p>$TIMESTAMP</p>
              $(report_uptime)
              $(report_disk_space)
              $(report_home_space)
         </body>
    </html>
    _EOF_
    return
}

# output html page

if [[ -n "$filename" ]]; then
    if touch "$filename" && [[ -f "$filename" ]]; then
         write_html_page > "$filename"
    else
         echo "$PROGNAME: Cannot write file '$filename'" >&2
         exit 1
    fi
else
    write_html_page
fi
```

The code that handles the logic of the `-f` option appears at the end of the previous listing. In it, we test for the existence of a filename and, if one is found, a test is performed to see whether the file is indeed writable. To do this, a `touch` is performed, followed by a test to determine whether the resulting file is a regular file. These two tests take care of situations where an invalid pathname is input (`touch` will fail), and, if the file already exists, that it's a regular file.

As we can see, the `write_html_page` function is called to perform the actual generation of the page. Its output is either directed to standard output (if the variable `filename` is empty) or redirected to the specified file. Since we have two possible destinations for the HTML code, it makes sense to convert the `write_html_page` routine to a shell function to avoid redundant code.

Summing Up

With the addition of positional parameters, we can now write fairly functional scripts. For simple, repetitive tasks, positional parameters make it possible to write very useful shell functions that can be placed in a user's `.bashrc` file.

Our `sys_info_page` program has grown in complexity and sophistication. Here is a complete listing, with the most recent changes highlighted:

```
#!/bin/bash

# sys_info_page: program to output a system information page

PROGNAME="$(basename "$0")"
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME="$(date +"%x %r %Z")"
TIMESTAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
    cat <<- _EOF_
         <h2>System Uptime</h2>
         <pre>$(uptime)</pre>
         _EOF_
    return
}

report_disk_space () {
    cat <<- _EOF_
         <h2>Disk Space Utilization</h2>
         <pre>$(df -h)</pre>
         _EOF_
    return
}
```

```
report_home_space () {
    if [[ "$(id -u)" -eq 0 ]]; then
        cat <<- _EOF_
            <h2>Home Space Utilization (All Users)</h2>
            <pre>$(du -sh /home/*)</pre>
            _EOF_
    else
        cat <<- _EOF_
            <h2>Home Space Utilization ($USER)</h2>
            <pre>$(du -sh "$HOME")</pre>
            _EOF_
    fi
    return
}

usage () {
    echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
    return
}

write_html_page () {
    cat <<- _EOF_
    <html>
        <head>
            <title>$TITLE</title>
        </head>
        <body>
            <h1>$TITLE</h1>
            <p>$TIMESTAMP</p>
            $(report_uptime)
            $(report_disk_space)
            $(report_home_space)
        </body>
    </html>
    _EOF_
    return
}

# process command line options

interactive=
filename=
```

```
while [[ -n "$1" ]]; do
    case "$1" in
        -f | --file)        shift
                            filename="$1"
                            ;;
        -i | --interactive) interactive=1
                            ;;
        -h | --help)        usage
                            exit
                            ;;
        *)                  usage >&2
                            exit 1
                            ;;
    esac
    shift
done

# interactive mode

if [[ -n "$interactive" ]]; then
    while true; do
        read -p "Enter name of output file: " filename
        if [[ -e "$filename" ]]; then
            read -p "'$filename' exists. Overwrite? [y/n/q] > "
            case "$REPLY" in
                Y|y) break
                     ;;
                Q|q) echo "Program terminated."
                     exit
                     ;;
                *)   continue
                     ;;
            esac
        elif [[ -z "$filename" ]]; then
            continue
        else
            break
        fi
    done
fi

# output html page

if [[ -n "$filename" ]]; then
```

```
    if touch "$filename" && [[ -f "$filename" ]]; then
         write_html_page > "$filename"
    else
         echo "$PROGNAME: Cannot write file '$filename'" >&2
         exit 1
    fi
else
    write_html_page
fi
```

We're not done yet. There are still a few more things we can do and improvements we can make.

Further Reading

- The *Bash Hackers Wiki* has a good article on positional parameters:
  http://wiki.bash-hackers.org/scripting/posparams
- The *Bash Reference Manual* has an article on the special parameters, including `$*` and `$@`:
  http://www.gnu.org/software/bash/manual/bashref.html#Special-Parameters
- In addition to the techniques discussed in this chapter, `bash` includes a builtin command called `getopts`, which can also be used for process command line arguments. It is described in the SHELL BUILTIN COMMANDS section of the `bash` man page and at the *Bash Hackers Wiki*:
  http://wiki.bash-hackers.org/howto/getopts_tutorial