Positional Parameters

When we last left our script, it looked something like this:

```
#!/bin/bash

# sysinfo_page - A script to produce a system information HTML file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW="$(date +"%x %r %Z")"
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Functions

system_info()
{
    echo "<h2>System release info</h2>"
    echo "<p>Function not yet implemented</p>"

}   # end of system_info


show_uptime()
{
    echo "<h2>System uptime</h2>"
    echo "<pre>"
    uptime
    echo "</pre>"

}   # end of show_uptime


drive_space()
{
    echo "<h2>Filesystem space</h2>"
    echo "<pre>"
    df
    echo "</pre>"

}   # end of drive_space


home_space()
{
    # Only the superuser can get this information
    if [ "$(id -u)" = "0" ]; then
        echo "<h2>Home directory space by user</h2>"
        echo "<pre>"
        echo "Bytes Directory"
        du -s /home/* | sort -nr
        echo "</pre>"
    fi

}   # end of home_space



##### Main

cat <<- _EOF_
  <html>
  <head>
      <title>$TITLE</title>
  </head>
  <body>
      <h1>$TITLE</h1>
      <p>$TIME_STAMP</p>
      $(system_info)
      $(show_uptime)
      $(drive_space)
      $(home_space)
  </body>
  </html>
_EOF_
```

We have most things working, but there are several more features we can add:

1. We should be able to specify the name of the output file on the command line, as well as set a default output file name if no name is specified.

2. Let's offer an interactive mode that will prompt for a file name and warn the user if the file exists and prompt the user to overwrite it.

3. Naturally, we want to have a help option that will display a usage message.

All of these features involve using command line options and arguments. To handle options on the command line, we use a facility in the shell called *positional parameters*. Positional parameters are a series of special variables (`$0` through `$9`) that contain the contents of the command line.

Let's imagine the following command line:

```
[me@linuxbox me]$ some_program word1 word2 word3
```

If `some_program` were a bash shell script, we could read each item on the command line because the positional parameters contain the following:

- $0 would contain "some_program"
- $1 would contain "word1"

- $1 would contain "word1"
- $2 would contain "word2"
- $3 would contain "word3"

Here is a script we can use to try this out:

```
#!/bin/bash

echo "Positional Parameters"
echo '$0 = ' $0
echo '$1 = ' $1
echo '$2 = ' $2
echo '$3 = ' $3
```

Detecting Command Line Arguments

Often, we will want to check to see if we have comand line arguments on which to act. There are a couple of ways to do this. First, we could simply check to see if $1 contains anything like so:

```
#!/bin/bash

if [ "$1" != "" ]; then
    echo "Positional parameter 1 contains something"
else
    echo "Positional parameter 1 is empty"
fi
```

Second, the shell maintains a variable called $# that contains the number of items on the command line in addition to the name of the command ($0).

```
#!/bin/bash

if [ $# -gt 0 ]; then
    echo "Your command line contains $# arguments"
else
    echo "Your command line contains no arguments"
fi
```

Command Line Options

As we discussed before, many programs, particularly ones from the GNU Project, support both short and long command line options. For example, to display a help message for many of these programs, we may use either the "-h" option or the longer "--help" option. Long option names are typically preceded by a double dash. We will adopt this convention for our scripts.

Here is the code we will use to process our command line:

```
interactive=
filename=~/sysinfo_page.html

while [ "$1" != "" ]; do
```

```
    case $1 in
        -f | --file )           shift
                                filename="$1"
                                ;;
        -i | --interactive )    interactive=1

                                ;;
        -h | --help )           usage
                                exit
                                ;;
        * )                     usage
                                exit 1
    esac
    shift
done
```

This code is a little tricky, so we need to explain it.

The first two lines are pretty easy. We set the variable `interactive` to be empty. This will indicate that the interactive mode has not been requested. Then we set the variable `filename` to contain a default file name. If nothing else is specified on the command line, this file name will be used.

After these two variables are set, we have default settings, in case the user does not specify any options.

Next, we construct a **`while`** loop that will cycle through all the items on the command line and process each one with **`case`**. The **`case`** will detect each possible option and process it accordingly.

Now the tricky part. How does that loop work? It relies on the magic of **`shift`**.

`shift` is a shell builtin that operates on the positional parameters. Each time we invoke **`shift`**, it "shifts" all the positional parameters down by one. `$2` becomes `$1`, `$3` becomes `$2`, `$4` becomes `$3`, and so on. Try this:

```
#!/bin/bash

echo "You start with $# positional parameters"

# Loop until all parameters are used up
while [ "$1" != "" ]; do
    echo "Parameter 1 equals $1"
    echo "You now have $# positional parameters"

    # Shift all the parameters down by one
    shift

done
```

Getting an Option's Argument

Our "`-f`" option requires a valid file name as an argument. We use **`shift`** again to get the next item from the command line and assign it to `filename`. Later we will have to check the content of `filename` to make sure it is valid.

Integrating the Command Line Processor into the Script

We will have to move a few things around and add a usage function to get this new routine integrated into our script. We'll also add some test code to verify that the command line processor is working correctly. Our script now looks like this:

```
#!/bin/bash

# sysinfo_page - A script to produce a system information HTML file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW="$(date +"%x %r %Z")"
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Functions

system_info()
{
    echo "<h2>System release info</h2>"
    echo "<p>Function not yet implemented</p>"

}   # end of system_info


show_uptime()
{
    echo "<h2>System uptime</h2>"
    echo "<pre>"
    uptime
    echo "</pre>"

}   # end of show_uptime


drive_space()
{
    echo "<h2>Filesystem space</h2>"
    echo "<pre>"
    df
    echo "</pre>"

}   # end of drive_space


home_space()
{
    # Only the superuser can get this information

    if [ "$(id -u)" = "0" ]; then
        echo "<h2>Home directory space by user</h2>"
        echo "<pre>"
        echo "<pre>"
        echo "Bytes Directory"
        du -s /home/* | sort -nr
        echo "</pre>"
    fi


}   # end of home_space


write_page()
{
    cat <<- _EOF_
    <html>
        <head>
        <title>$TITLE</title>
        </head>
        <body>
        <h1>$TITLE</h1>
        <p>$TIME_STAMP</p>
        $(system_info)
        $(show_uptime)
        $(drive_space)
        $(home_space)
        </body>
    </html>
_EOF_

}

usage()
{
    echo "usage: sysinfo_page [[[-f file ] [-i]] | [-h]]"
}


##### Main

interactive=
filename=~/sysinfo_page.html

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

```
# Test code to verify command line processing

if [ "$interactive" = "1" ]; then
  echo "interactive is on"
else
  echo "interactive is off"
fi
echo "output file = $filename"


# Write page (comment out until testing is complete)

# write_page > $filename
```

Adding Interactive Mode

The interactive mode is implemented with the following code:

```
if [ "$interactive" = "1" ]; then

    response=

    read -p "Enter name of output file [$filename] > " response
    if [ -n "$response" ]; then
        filename="$response"
    fi

    if [ -f $filename ]; then
        echo -n "Output file exists. Overwrite? (y/n) > "
        read response
        if [ "$response" != "y" ]; then
            echo "Exiting program."
            exit 1
        fi
    fi
fi
```

First, we check if the interactive mode is on, otherwise we don't have anything to do. Next, we ask the user for the file name. Notice the way the prompt is worded:

```
"Enter name of output file [$filename] > "
```

We display the current value of `filename` since, the way this routine is coded, if the user just presses the enter key, the default value of `filename` will be used. This is accomplished in the next two lines where the value of `response` is checked. If `response` is not empty, then `filename` is assigned the value of `response`. Otherwise, `filename` is left unchanged, preserving its default value.

After we have the name of the output file, we check if it already exists. If it does, we prompt the user. If the user response is not "y," we give up and exit, otherwise we can proceed.