Search

Adding arguments and options to your Bash scripts

Exploring methods for getting data into scripts and controlling the script's execution path for better automation and script management.

Posted: May 19, 2021 | 14 min read | David Both

Image by Gerd Altmann from Pixabay

One of the most important tools for most sysadmins is automation. We write and maintain scripts to automate the common and frequent tasks that we must perform.

More Linux resources

- Advanced Linux commands cheat sheet
- Download RHEL 9 for free through the Red Hat Developer program
- A guide to installing

I have dozens of scripts—short and long—that I've written and modified over the years. Some of my most useful scripts have been to perform regular backups early each morning, install updated software packages with fixes and enhancements, and upgrade from one version of Fedora to the next. I just upgraded all of my personal hosts and servers to Fedora 34 a few days ago using a fairly simple script.

Two of the most common things I do for all my scripts are creating a help function and a function that displays the GPL3 license statement. I like to include verbose or test modes to assist in problem determination in my scripts. In some scripts, I also pass values such as a user name, the version of Fedora to upgrade to, file names, and more.

The ability to use positional parameters—otherwise known as arguments—to specify data to be used as values for variables in the scripts is one method for accomplishing this. Another is the use of options and option arguments. This article explores these two methods for getting data into the script and controlling the script's execution path.

Positional parameters

Bash uses a tool called positional parameters to provide a means of entering data into a Bash program when it is invoked from the command line. There are ten positional parameters that run from **$0** through **$9**, although there are ways to hack around that limit.

Starting with a simple script that displays an entered name on the screen. Create a file called `script1.sh` with the following content and make it executable.

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

I placed this script in my `~/bin directory`, where personal executable files such as scripts are intended to be stored. Look at your **$PATH** variable, which contains `/home/username/bin` as one component. If the `~/bin` directory does not exist, you can create it. Or you can just put this file wherever you want and use it from there.

Then run the script with no parameters.

```
[student@testvm1 ~]$ script1.sh
/home/dboth/bin/script1.sh
[student@testvm1 ~]$
```

The output from this script is the name of the script. The **$0** parameter is reserved and predefined as the name of the running script and cannot be used for any other purpose. This can be handy inside a script because you don't need to pass the script its own name if it requires it.

So change the script to use **$1** for the positional variable, and run it again:

```
#!/bin/bash
echo $1
```

Run it again, this time using a single parameter:

```
[student@testvm1 ~]$ script1.sh help
```

What happens if the parameter is two words?

```
[student@testvm1 ~]$ script1.sh help me
help
[student@testvm1 ~]$
```

That is actually two parameters, but you can remedy that with quotes, as seen here:

```
[student@testvm1 ~]$ script1.sh "help me"
help me
[student@testvm1 ~]$
```

This can be helpful where the input is supposed to be a street address or something with multiple words, like this:

```
[student@testvm1 ~]$ script1.sh "80486 Intel St."
80486 Intel St.
[student@testvm1 ~]$
```

But there are times when you do need multiple parameters, such as with names or full addresses.

[You might also like: More stupid Bash tricks: Variables, find, file descriptors, and remote operations]

Change the program to look like this:

```
#!/bin/bash
echo "Name: $1"
echo "Street: $2"
echo "City: $3"
echo "State/Province/Territory: $4"
echo "Zip/Postal code: $5"
```

And run it using the parameters as shown:

```
[student@testvm1 ~]$ script1.sh "David Both" "80486 Intel St." Raleigh NC XXXXX
Name: David Both
Street: 80486 Intel St.
City: Raleigh
State/Province/Territory: NC
Zip/Postal code: XXXXX
```

Of course, there are many ways to use the positional parameters once values have been assigned, but this little program makes it easy to see what is happening. It also makes it easy to experiment in a safe way.

[Free download: Advanced Linux commands cheat sheet.]

Try putting the parameters in a different order to see how that works. These parameters are positional, and that is a key consideration. You must consider how many parameters are needed, how the user remembers them, and what order to place them.

You need a way to make the order of the parameters irrelevant and still need a way to modify the execution path.

center

- Implementing DevSecOps guide
- Red Hat CVE checker

Options

You can do those two things using command line options.

I find that even simple Bash programs should have some sort of help facility, even if it is fairly rudimentary. Many of the Bash shell programs I write are used infrequently enough that I may forget the exact syntax of the command I need to issue. Some are just so complex that I need to review the options and arguments required even though I use them frequently.

Having a built-in help function allows you to view those things without resorting to inspecting the code itself. A good and complete help facility is also one part of program documentation.

About functions

Shell functions are lists of Bash program statements stored in the shell's environment and can be executed like any other command by typing its name at the command line. Shell functions may also be known as procedures or subroutines, depending upon which other programming language you might be using.

[Get this free Bash shell scripting cheat sheet.]

Functions are called in your scripts or from the CLI by using their names, just as you would for any other command. In a CLI program or a script, the commands in the function are executed when called. Then the sequence of program flow returns to the calling entity, and the next series of program statements in that entity is executed.

The syntax of a function is:

```
FunctionName(){program statements}
```

Create a simple function at the CLI. The function is stored in the shell environment for the shell instance in which it is created. You're going to create a function called **hw**, which stands for *Hello world*. Enter the following code at the CLI and press **Enter**. Then enter **hw** as you would any other shell command.

```
[student@testvm1 ~]$ hw(){ echo "Hi there kiddo"; }
[student@testvm1 ~]$ hw
Hi there kiddo
[student@testvm1 ~]$
```

Ok, so I am a little tired of the standard "Hello world!" I usually start with. Now list all of the currently defined functions. There are a lot of them, so I have shown just the new **hw** function. When called from the command line or within a program, a function performs its programmed task. It then exits, returning control to the calling entity, the command line, or the next Bash program statement in a script after the calling statement.

```
[student@testvm1 ~]$ declare -f | less
<snip>
hw ()
{
    echo "Hi there kiddo"
}
<snip>
```

```
[student@testvm1 ~]$ unset -f hw ; hw
bash: hw: command not found
[student@testvm1 ~]$
```

The hello.sh script

Create a new Bash shell script, `~/bin/hello.sh`, and make it executable. Add the following content, keeping it basic to start:

```
#!/bin/bash
echo "hello world!"
```

Run it to verify that it prints "hello world!"

```
[dboth@david ~]$ hello.sh
hello world!
[dboth@david ~]$
```

I know—I can't help myself, so I went back to "hello world!".

Creating the help function

Add the **help** function shown below to the code of the hello program. Place the **help** function between the two statements you already have. This **help** function will display a short description of the program, a syntax diagram, and a short description of each available option. You also add a call to the **help** function to test it and some comment lines that provide a visual demarcation between the functions and the main portion of the program.

The program now looks like this.

```
#!/bin/bash
############################################################
# Help                                                     #
############################################################
Help()
{
   # Display Help
   echo "Add description of the script functions here."
   echo
   echo "Syntax: scriptTemplate [-g|h|v|V]"
   echo "options:"
   echo "g     Print the GPL license notification."
   echo "h     Print this Help."
   echo "v     Verbose mode."
   echo "V     Print software version and exit."
   echo
}

############################################################
############################################################
# Main program                                             #
############################################################
############################################################

Help
echo "Hello world!"
```

The options described in this **help** function might be typical in the programs I write, although none are yet present in the code. Run the program to test it.

```
[student@testvm1 ~]$ hello.sh
Add a description of the script functions here.

Syntax: scriptTemplate [-g|h|v|V]
options:
g     Print the GPL license notification.
h     Print this Help.
v     Verbose mode.
V     Print software version and exit.

Hello world!
[student@testvm1 ~]$
```

Because you haven't added any logic to display the help when you want it, the program will always display the help. However, you know that the function is working correctly, so you can add some logic only to show the help when you use a `-h` option at the command line invocation of the program.

[Want to test your sysadmin skills? Take a skills assessment today.]

Handling options

The ability for a Bash script to handle command line options such as `-h` to display help gives you some powerful capabilities to direct the program and modify what it does. In the case of your `-h` option, you want the program to print the help text to the terminal session and then quit without running the rest of the program. The ability to process options entered at the command line can be added to the Bash script using the `while` command in conjunction with the `getops` and `case` commands.

The `getops` command reads any and all options specified at the command line and creates a list of those options. The `while` command loops through the list of options by setting the variable **$options** for each in the code below. The `case` statement is used to evaluate each option in turn and execute the statements in the corresponding stanza. The `while` statement will continue to assess the list of options until they have all been processed or an exit statement is encountered, which terminates the program.

Be sure to delete the **help** function call just before the **echo "Hello world!"** statement so that the main body of the program now looks like this.

```
############################################################
############################################################
# Main program                                             #
############################################################
############################################################
############################################################
# Process the input options. Add options as needed.        #
############################################################
# Get the options
while getopts ":h" option; do
   case $option in
      h) # display Help
         Help
         exit;;
   esac
done

echo "Hello world!"
```

Notice the double semicolon at the end of the exit statement in the case option for `-h`. This is required for each option. Add to this case statement to delineate the end of each option.

```
[student@testvm1 ~]$ hello.sh
Hello world!
```

That works, so now test the logic that displays the help text.

```
[student@testvm1 ~]$ hello.sh -h
Add a description of the script functions here.

Syntax: scriptTemplate [-g|h|t|v|V]
options:
g     Print the GPL license notification.
h     Print this Help.
v     Verbose mode.
V     Print software version and exit.
```

That works as expected, so now try some testing to see what happens when you enter some unexpected options.

```
[student@testvm1 ~]$ hello.sh -x
Hello world!
```

```
[student@testvm1 ~]$ hello.sh -q
Hello world!
```

```
[student@testvm1 ~]$ hello.sh -lkjsahdf
Add a description of the script functions here.

Syntax: scriptTemplate [-g|h|t|v|V]
options:
g     Print the GPL license notification.
h     Print this Help.
v     Verbose mode.
V     Print software version and exit.

[student@testvm1 ~]$
```

Handling invalid options

The program just ignores the options for which you haven't created specific responses without generating any errors. Although in the last entry with the `-lkjsahdf` options, because there is an "h" in the list, the program did recognize it and print the help text. Testing has shown that one thing that is missing is the ability to handle incorrect input and terminate the program if any is detected.

You can add another case stanza to the case statement that will match any option for which there is no explicit match. This general case will match anything you haven't provided a specific match for. The *case* statement now looks like this.

```
while getopts ":h" option; do
   case $option in
      h) # display Help
         Help
         exit;;
     \?) # Invalid option
         echo "Error: Invalid option"
         exit;;
   esac
done
```

Kubernetes and OpenShift

- Kubernetes cheat sheet
- Interactive course: Getting started with OpenShift
- Red Hat OpenShift and Kubernetes ... what's the difference?
- Interactive course: Create a cluster in Red Hat OpenShift Service on AWS with S...
- Get started with Red Hat OpenShift Service on AWS

This bit of code deserves an explanation about how it works. It seems complex but is fairly easy to understand. The **while – done** structure defines a loop that executes once for each option in the **getopts – option** structure. The **":h"** string —which requires the quotes—lists the possible input options that will be evaluated by the **case – esac** structure. Each option listed must have a corresponding stanza in the case statement. In this case, there are two. One is the **h)** stanza which calls the Help procedure. After the Help procedure completes, execution returns to the next program statement, **exit;;** which exits from the program without executing any more code even if some exists. The option processing loop is also terminated, so no additional options would be checked.

Notice the catch-all match of **\?** as the last stanza in the case statement. If any options are entered that are not recognized, this stanza prints a short error message and exits from the program.

Any additional specific cases must precede the final catch-all. I like to place the case stanzas in alphabetical order, but there will be circumstances where you want to ensure that a particular case is processed before certain other ones. The case statement is sequence sensitive, so be aware of that when you construct yours.

The last statement of each stanza in the case construct must end with the double semicolon (; ;), which is used to mark the end of each stanza explicitly. This allows those programmers who like to use explicit semicolons for the end of each statement instead of implicit ones to continue to do so for each statement within each case stanza.

Test the program again using the same options as before and see how this works now.

The Bash script now looks like this.

```
#!/bin/bash
############################################################
# Help                                                     #
############################################################
Help()
{
   # Display Help
   echo "Add description of the script functions here."
   echo
   echo "Syntax: scriptTemplate [-g|h|v|V]"
   echo "options:"
   echo "g     Print the GPL license notification."
   echo "h     Print this Help."
   echo "v     Verbose mode."
   echo "V     Print software version and exit."
   echo
}

############################################################
############################################################
# Main program                                             #
############################################################
############################################################
############################################################
# Process the input options. Add options as needed.        #
############################################################
# Get the options
while getopts ":h" option; do
   case $option in
      h) # display Help
         Help
         exit;;
     \?) # Invalid option
         echo "Error: Invalid option"
         exit;;
   esac
done


echo "hello world!"
```

Be sure to test this version of your program very thoroughly. Use random input and see what happens. You should also try testing valid and invalid options without using the dash (`-`) in front.

Using options to enter data

First, add a variable and initialize it. Add the two lines shown in bold in the segment of the program shown below. This initializes the **$Name** variable to "world" as the default.

```
<snip>
############################################################
############################################################
# Main program                                             #
############################################################
############################################################

# Set variables
Name="world"

############################################################
# Process the input options. Add options as needed.        #
```

Change the last line of the program, the `echo` command, to this.

```
echo "hello $Name!"
```

Add the logic to input a name in a moment but first test the program again. The result should be exactly the same as before.

```
[dboth@david ~]$ hello.sh
hello world!
[dboth@david ~]$
```

```
# Get the options
while getopts ":hn:" option; do
   case $option in
      h) # display Help
         Help
         exit;;
      n) # Enter a name
         Name=$OPTARG;;
     \?) # Invalid option
         echo "Error: Invalid option"
         exit;;
   esac
done
```

$OPTARG is always the variable name used for each new option argument, no matter how many there are. You must assign the value in **$OPTARG** to a variable name that will be used in the rest of the program. This new stanza does not have an exit statement. This changes the program flow so that after processing all valid options in the case statement, execution moves on to the next statement after the case construct.

Test the revised program.

```
[dboth@david ~]$ hello.sh
hello world!
```

```
[dboth@david ~]$ hello.sh -n LinuxGeek46
hello LinuxGeek46!
```

```
[dboth@david ~]$ hello.sh -n "David Both"
hello David Both!
[dboth@david ~]$
```

The completed program looks like this.

```
#!/bin/bash
############################################################
# Help                                                     #
############################################################
Help()
{
   # Display Help
   echo "Add description of the script functions here."
   echo
   echo "Syntax: scriptTemplate [-g|h|v|V]"
   echo "options:"
   echo "g     Print the GPL license notification."
   echo "h     Print this Help."
   echo "v     Verbose mode."
   echo "V     Print software version and exit."
   echo
}

############################################################
############################################################
# Main program                                             #
############################################################
############################################################

# Set variables
Name="world"

############################################################
# Process the input options. Add options as needed.        #
############################################################
# Get the options
while getopts ":hn:" option; do
   case $option in
      h) # display Help
         Help
         exit;;
      n) # Enter a name
         Name=$OPTARG;;
     \?) # Invalid option
         echo "Error: Invalid option"
         exit;;
   esac
done


echo "hello $Name!"
```

Be sure to test the help facility and how the program reacts to invalid input to verify that its ability to process those has not been compromised. If that all works as it should, then you have successfully learned how to use options and option arguments.

Wrap up

exams FAQ

- Get essential IT career advice from IT leaders

In this article, you've used positional parameters to enter data into the Bash program during invocation from the command line and used options to direct the flow of the program as well as to enter data into the program. You added a help function and the ability to process command line options to display the help selectively. And you added an optional argument that allows entering a name on the command line.

This little test program is designed to be simple, so you can easily experiment with it yourself to test this input method on your own. As an exercise, revise the program to take a first name and last name. Try entering the options for first and last names in reverse order to see what happens.

Resources

- How to program with Bash: Syntax and tools
- How to program with Bash: Logical operators and shell expansions
- How to program with Bash: Loops

Check out these related articles on Enable Sysadmin

Using Bash for automation

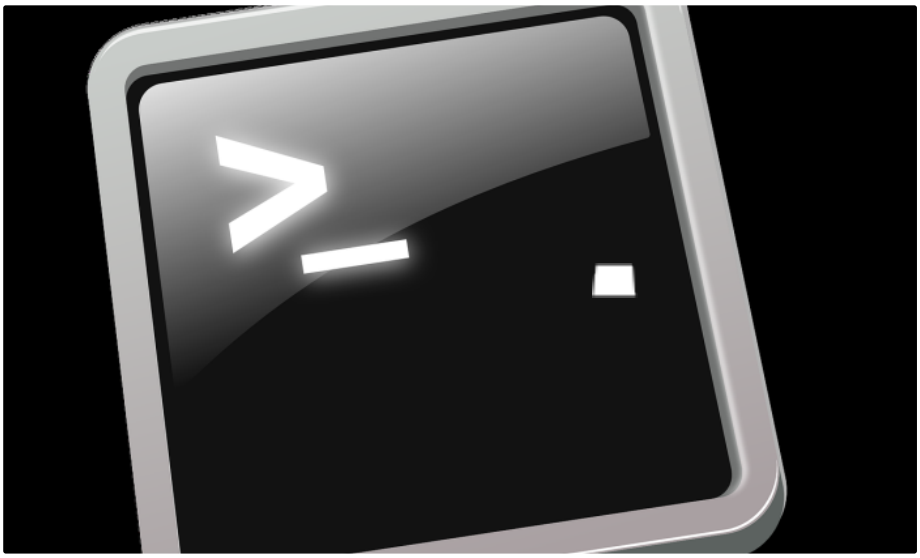
Take your Bash scripting to a new higher level using libraries with automation.

Posted: February 13, 2020
Author: Chris Evich (Red Hat)

Bash command line exit codes demystified

If you've ever wondered what an exit code is or why it's a 0, 1, 2, or even 255, you're in the right place.

Posted: February 4, 2020
Author: Ken Hess (Red Hat)



Parsing Bash history in Linux

The history command isn't always about reducing key presses. Find out how you can leverage command history into more efficient system administration.

Posted: March 30, 2020
Author: Seth Kenlon (Red Hat)

Topics: Linux | Linux administration | Scripting | Command line utilities

David Both

David Both is an Open Source Software and GNU/Linux advocate, trainer, writer, and speaker who lives in Raleigh North Carolina. He is a strong proponent of and evangelist for the "Linux Philosophy." More about me

Related Content

Linux tool alternatives: 6 replacements for traditional favorites

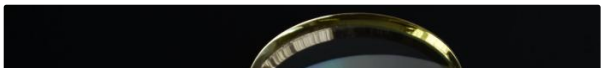
Consider swapping Linux tools for these alternatives that provide more features and functionality.

Posted: July 27, 2022
Author: Jose Vicente Nunez (Sudoer)

How to hide PID listings from non-root users in Linux

Prevent average users from viewing your Linux system's processes with the hidepid command.

Posted: July 26, 2022
Author: Emad Al-Mousa



d in the United States and other countries.

Privacy Policy
Terms of Use
All policies and guidelines

Red Hat Summit

How to troubleshoot your network with Nmap

Learn how to use Nmap scans to check if systems are online and find problems in your network.

Posted: July 13, 2022
Author: Gabrielle Stenzel

OUR BEST CONTENT, DELIVERED TO YOUR INBOX

Enter your email address...

Select your country or region

Subscribe

Privacy Statement