A sysadmin's guide to Bash scripting

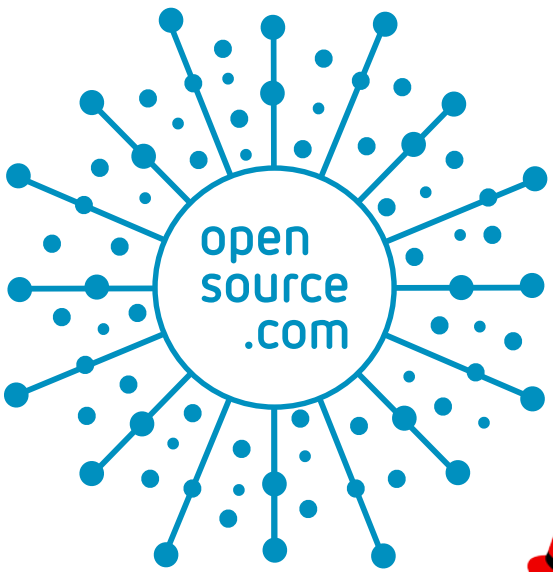
ABOUT OPENSOURCE.COM

What is Opensource.com?

OPENSOURCE.COM publishes stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Submit a story idea: opensource.com/story

Email us: open@opensource.com



Supported by
Red Hat

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."

David has been in the IT industry for nearly 50 years. He has taught RHCE classes for Red Hat and has worked at MCI Worldcom, Cisco, and the State of North Carolina. He has been working with Linux and Open Source Software for over 20 years.

David prefers to purchase the components and build his own computers from scratch to ensure that each new computer meets his exacting specifications. His primary workstation is an ASUS TUF X299 motherboard and an Intel i9 CPU with 16 cores (32 CPUs) and 64GB of RAM in a ThermalTake Core X9 case.

David has written articles for magazines including, Linux Magazine and Linux Journal. His article "Complete Kickstart," co-authored with a colleague at Cisco, was ranked 9th in the Linux Magazine Top Ten Best System Administration Articles list for 2008. David currently writes prolifically for OpenSource.com and Enable SysAdmin.

He currently has four books published at Apress, "The Linux Philosophy for SysAdmins," and "Using and Administering Linux: Zero to SysAdmin," a Linux self-study training course in three volumes.

FOLLOW DAVID BOTH

Email: LinuxGeek46@both.org

Twitter: @LinuxGeek46

CHAPTERS

"This guide is partially based on Volume 2, Chapter 10 of David Both's three-part Linux self-study course, Using and Administering Linux—Zero to SysAdmin."

Introduction to automation with Bash scripts

In the first part in this four-part guide, learn how to create a simple shell script and why they are the best way to automate tasks.

SYSADMINS, those of us who run and manage Linux computers most closely, have direct access to tools that help us work more efficiently. To help you use these tools to their maximum benefit to make your life easier, this guide explores using automation in the form of Bash shell scripts. It covers:

- The advantages of automation with Bash shell scripts
- Why using shell scripts is a better choice for sysadmins than compiled languages like C or C++
- Creating a set of requirements for new scripts
- Creating simple Bash shell scripts from command-line interface (CLI) programs
- Enhancing security through using the user ID (UID) running the script
- Using logical comparison tools to provide execution flow control for command-line programs and scripts
- Using command-line options to control script functionality
- Creating Bash functions that can be called from one or more locations within a script
- Why and how to license your code as open source
- Creating and implementing a simple test plan

I previously wrote a series of articles about Bash commands and syntax and creating Bash programs at the command line, which you can find in the references section at the end of this part. But this four part guide is as much about creating scripts (and some techniques that I find useful) as it is about Bash commands and syntax.

Why I use shell scripts

In Chapter 9 of The Linux Philosophy for Sysadmins [1], I write:

> "A sysadmin is most productive when thinking—thinking about how to solve existing problems and about how to avoid future problems; thinking about how to monitor Linux computers in order to find clues that anticipate and foreshadow those future problems; thinking about how to make [their] job more efficient; thinking about how to automate all of those tasks that need to be performed whether every day or once a year."

> "Sysadmins are next most productive when creating the shell programs that automate the solutions that they have conceived while appearing to be unproductive. The more automation we have in place, the more time we have available to fix real problems when they occur and to contemplate how to automate even more than we already have."

This first part explores why shell scripts are an important tool for the sysadmin and the basics of creating a very simple Bash script.

Why automate?

Have you ever performed a long and complex task at the command line and thought, "Glad that's done. Now I never have to worry about it again!"? I have—frequently. I ultimately figured out that almost everything that I ever need to do on a computer (whether mine or one that belongs to an employer or a consulting customer) will need to be done again sometime in the future.

Of course, I always think that I will remember how I did the task. But, often, the next time is far enough into the future that I forget that I have ever done it, let alone how to do it. I started writing down the steps required for some tasks on bits of paper, then thought, "How stupid of me!" So I transferred those scribbles to a simple notepad application on my computer, until one day, I thought again, "How stupid of me!" If I am going to store this data on my computer, I might as well create a shell script and store it in a standard location, like **/usr/local/bin** or **~/bin**, so I can just type the name of the shell program and let it do all the tasks I used to do manually.

For me, automation also means that I don't have to remember or recreate the details of how I performed the task in order to do it again. It takes time to remember how to do things and time to type in all the commands. This can become a significant time sink for tasks that require typing large numbers of long commands. Automating tasks by creating shell scripts reduces the typing necessary to perform routine tasks.

Shell scripts

Writing shell programs—also known as scripts—is the best strategy for leveraging my time. Once I write a shell program,

I can rerun it as many times as I need to. I can also update my shell scripts to compensate for changes from one release of Linux to the next, installing new hardware and software, changing what I want or need to accomplish with the script, adding new functions, removing functions that are no longer needed, and fixing the not-so-rare bugs in my scripts. These kinds of changes are just part of the maintenance cycle for any type of code.

Every task performed via the keyboard in a terminal session by entering and executing shell commands can and should be automated. Sysadmins should automate everything we are asked to do or decide needs to be done. Many times, doing the automation upfront saves me time the first time.

One Bash script can contain anywhere from a few commands to many thousands. I have written Bash scripts with only one or two commands, and I have written a script with over 2,700 lines, more than half of which are comments.

Getting started

Here's a trivial example of a shell script and how to create it. In my earlier guide on Bash command-line programming, I used the example from every book on programming I have ever read: "Hello world." From the command line, it looks like this:

```
[student@testvm1 ~]$ echo "Hello world"
Hello world
```

By definition, a program or shell script is a sequence of instructions for the computer to execute. But typing them into the command line every time is quite tedious, especially when the programs are long and complex. Storing them in a file that can be executed with a single command saves time and reduces the possibility for errors to creep in.

I recommend trying the following examples as a non-root user on a test system or virtual machine (VM). Although the examples are harmless, mistakes do happen, and being safe is always wise.

The first task is to create a file to contain your program. Use the touch command to create the empty file, **hello**, then make it executable:

```
[student@testvm1 ~]$ touch hello
[student@testvm1 ~]$ chmod 774 hello
```

Now, use your favorite editor to add the following line to the file:

```
echo "Hello world"
```

Save the file and run it from the command line. You can use a separate shell session to execute the scripts in this guide:

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

This is the simplest Bash program you may ever create—a single statement in a file. For this exercise, your complete shell script will be built around this simple Bash statement. The function of the program is irrelevant for this purpose, and this simple statement allows you to build a program structure—a template for other programs—without being concerned about the logic of a functional purpose. You can concentrate on the basic program structure and creating your template in a very simple way, and you can create and test the template itself rather than a complex functional program.

Shebang

The single statement works fine as long as you use Bash or a shell compatible with the commands used in the script. If no shell is specified in the script, the default shell will be used to execute the script commands.

The next task is to ensure that the script will run using the Bash shell, even if another shell is the default. This is accomplished with the shebang line. Shebang is the geeky way to describe the **#!** characters that explicitly specify which shell to use when running the script. In this case, that is Bash, but it could be any other shell. If the specified shell is not installed, the script will not run.

Add the shebang line as the first line of the script, so now it looks like this:

```
#!/usr/bin/bash
echo "Hello world!"
```

Run the script again—you should see no difference in the result. If you have other shells installed (such as ksh, csh, tcsh, zsh, etc.), start one and run the script again.

Scripts vs. compiled programs

When writing programs to automate—well, everything—sysadmins should always use shell scripts. Because shell scripts are stored in ASCII text format, they can be viewed and modified by humans just as easily as they can by computers. You can examine a shell program and see exactly what it does and whether there are any obvious errors in the syntax or logic. This is a powerful example of what it means to be *open*.

I know some developers consider shell scripts something less than "true" programming. This marginalization of shell scripts and those who write them seems to be predicated on the idea that the only "true" programming language is one that must be compiled from source code to produce executable code. I can tell you from experience that this is categorically untrue.

I have used many languages, including BASIC, C, C++, Pascal, Perl, Tcl/Expect, REXX (and some of its variations,

including Object REXX), many shell languages (including Korn, csh and Bash), and even some assembly language. Every computer language ever devised has had one purpose: to allow humans to tell computers what to do. When you write a program, regardless of the language you choose, you are giving the computer instructions to perform specific tasks in a specific sequence.

Scripts can be written and tested far more quickly than compiled languages. Programs usually must be written quickly to meet time constraints imposed by circumstances or the pointy-haired boss. Most scripts that sysadmins write are to fix a problem, to clean up the aftermath of a problem, or to deliver a program that must be operational long before a compiled program could be written and tested.

Writing a program quickly requires shell programming because it enables a quick response to the needs of the customer—whether that is you or someone else. If there are problems with the logic or bugs in the code, they can be corrected and retested almost immediately. If the original set of requirements is flawed or incomplete, shell scripts can be altered very quickly to meet the new requirements. In general, the need for speed of development in the sysadmin's job overrides the need to make the program run as fast as possible or to use as little as possible in the way of system resources like RAM.

Most things sysadmins do take longer to figure out how to do than to execute. Thus, it might seem counterproductive to create shell scripts for everything you do. It takes some time to write the scripts and make them into tools that produce reproducible results and can be used as many times as necessary. The time savings come every time you can run the script without having to figure out (again) how to do the task.

Final thoughts

This part didn't get very far with creating a shell script, but it did create a very small one. It also explored the reasons for creating shell scripts and why they are the most efficient option for the system administrator (rather than compiled programs).

In the next part, you will begin creating a Bash script template that can be used as a starting point for other Bash scripts. The template will ultimately contain a Help facility, a GNU licensing statement, a number of simple functions, and some logic to deal with those options, as well as others that might be needed for the scripts that will be based on this template.

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

Links

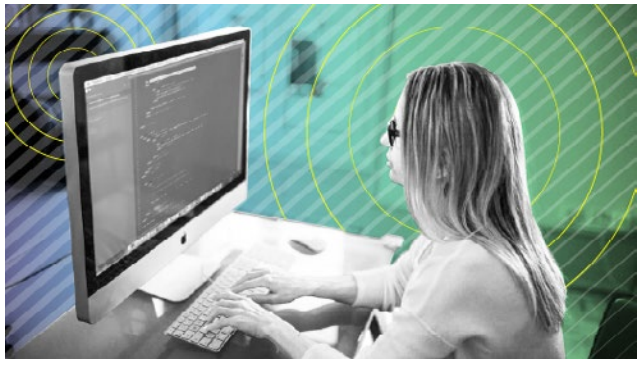
[1] http://www.both.org/?page_id=903

Creating a Bash script template

In the second part in this guide, create a fairly simple template that you can use as a starting point for other Bash programs, then test it.

IN THE FIRST PART in this guide, you created a very small, one-line Bash script and explored the reasons for creating shell scripts and why they are the most efficient option for the system administrator, rather than compiled programs.

In this second part, you will begin creating a Bash script template that can be used as a starting point for other Bash scripts. The template will ultimately contain a Help facility, a licensing statement, a number of simple functions, and some logic to deal with those options and others that might be needed for the scripts that will be based on this template.



Why create a template?

Like automation in general, the idea behind creating a template is to be the "lazy sysadmin [1]." A template contains the basic components that you want in all of your scripts. It saves time compared to adding those components to every new script and makes it easy to start a new script.

Although it can be tempting to just throw a few command-line Bash statements together into a file and make it executable, that can be counterproductive in the long run. A well-written and well-commented Bash program with a Help facility and the capability to accept command-line options provides a good starting point for sysadmins who maintain the program, which includes the programs that you write and maintain.

The requirements

You should always create a set of requirements for every project you do. This includes scripts, even if it is a simple list with only two or three items on it. I have been involved in many projects that either failed completely or failed to meet the customer's needs, usually due to the lack of a requirements statement or a poorly written one.

The requirements for this Bash template are pretty simple:

1. Create a template that can be used as the starting point for future Bash programming projects.
2. The template should follow standard Bash programming practices.
3. It must include:
   - A heading section that can be used to describe the function of the program and a changelog
   - A licensing statement
   - A section for functions
   - A Help function
   - A function to test whether the program user is root
   - A method for evaluating command-line options

The basic structure

A basic Bash script has three sections. Bash has no way to delineate sections, but the boundaries between the sections are implicit.

- All scripts must begin with the shebang (**#!**), and this must be the first line in any Bash program.
- The functions section must begin after the shebang and before the body of the program. As part of my need to document everything, I place a comment before each function with a short description of what it is intended to do. I also include comments inside the functions to elaborate further. Short, simple programs may not need functions.
- The main part of the program comes after the function section. This can be a single Bash statement or thousands of lines of code. One of my programs has a little over 200 lines of code, not counting comments. That same program has more than 600 comment lines.

That is all there is—just three sections in the structure of any Bash program.

Leading comments

I always add more than this for various reasons. First, I add a couple of sections of comments immediately after the shebang. These comment sections are optional, but I find them very helpful.

The first comment section is the program name and description and a change history. I learned this format while working at IBM, and it provides a method of documenting the long-term development of the program and any fixes applied to it. This is an important start in documenting your program.

The second comment section is a copyright and license statement. I use GPLv2, and this seems to be a standard statement for programs licensed under GPLv2. If you use a different open source license, that is fine, but I suggest add-

ing an explicit statement to the code to eliminate any possible confusion about licensing. Scott Peterson's article The source code is the license [2] helps explain the reasoning behind this.

So now the script looks like this:

```
#!/bin/bash
###############################################################
#                     scriptTemplate                          #
#                                                             #
# Use this template as the beginning of a new program. Place  #
# a short description of the script here.                     #
#                                                             #
# Change History                                              #
# 11/11/2019  David Both    Original code. This is a template #
#                           for creating new Bash shell       #
#                           scripts.                          #
#                           Add new history entries as needed.#
#                                                             #
#                                                             #
###############################################################
###############################################################
###############################################################
#                                                             #
#  Copyright (C) 2007, 2019 David Both                        #
#  LinuxGeek46@both.org                                       #
#                                                             #
#  This program is free software; you can redistribute it     #
#  and/or modify it under the terms of the GNU General Public #
#  License as published by the Free Software Foundation;      #
#  either version 2 of the License, or (at your option) any   #
#  later version.                                             #
#                                                             #
#  This program is distributed in the hope that it will be    #
#  useful, but WITHOUT ANY WARRANTY; without even the implied #
#  warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR    #
#  PURPOSE.  See the  GNU General Public License for more     #
#  details.                                                   #
#                                                             #
#  You should have received a copy of the GNU General Public  #
#  License along with this program; if not, write to the Free #
#  Software Foundation, Inc., 59 Temple Place, Suite 330,     #
#  Boston, MA  02111-1307  USA                                #
#                                                             #
###############################################################
###############################################################
###############################################################

echo "hello world!"
```

Run the revised program to verify that it still works as expected.

About testing

Now is a good time to talk about testing.

> "There is always one more bug."
> — Lubarsky's Law of Cybernetic Entomology

Lubarsky—whoever that might be—is correct. You can never find all the bugs in your code. For every bug I find, there always seems to be another that crops up, usually at a very inopportune time.

Testing is not just about programs. It is also about verification that problems—whether caused by hardware, software, or the seemingly endless ways users can find to break things—that are supposed to be resolved actually are. Just as important, testing is also about ensuring that the code is easy to use and the interface makes sense to the user.

Following a well-defined process when writing and testing shell scripts can contribute to consistent and high-quality results. My process is simple:

1. Create a simple test plan.
2. Start testing right at the beginning of development.
3. Perform a final test when the code is complete.
4. Move to production and test more.

The test plan

There are lots of different formats for test plans. I have worked with the full range—from having it all in my head; to a few notes jotted down on a sheet of paper; and all the way to a complex set of forms that require a full description of each test, which functional code it would test, what the test would accomplish, and what the inputs and results should be.

Speaking as a sysadmin who has been (but is not now) a tester, I try to take the middle ground. Having at least a short written test plan will ensure consistency from one test run to the next. How much detail you need depends upon how formal your development and test functions are.

The sample test plan documents I found using Google were complex and intended for large organizations with very formal development and test processes. Although those test plans would be good for people with "test" in their job title, they do not apply well to sysadmins' more chaotic and time-dependent working conditions. As in most other aspects of the job, sysadmins need to be creative. So here is a short list of things to consider including in your test plan. Modify it to suit your needs:

- The name and a short description of the software being tested
- A description of the software features to be tested
- The starting conditions for each test
- The functions to follow for each test
- A description of the desired outcome for each test
- Specific tests designed to test for negative outcomes
- Tests for how the program handles unexpected inputs
- A clear description of what constitutes pass or fail for each test
- Fuzzy testing, which is described below

This list should give you some ideas for creating your test plans. Most sysadmins should keep it simple and fairly informal.

Test early—test often

I always start testing my shell scripts as soon as I complete the first portion that is executable. This is true whether I am writing a short command-line program or a script that is an executable file.

I usually start creating new programs with the shell script template. I write the code for the Help function and test it. This is usually a trivial part of the process, but it helps me get started and ensures that things in the template are working properly at the outset. At this point, it is easy to fix problems with the template portions of the script or to modify it to meet needs that the standard template does not.

Once the template and Help function are working, I move on to creating the body of the program by adding comments to document the programming steps required to meet the program specifications. Now I start adding code to meet the requirements stated in each comment. This code will probably require adding variables that are initialized in that section of the template—which is now becoming a shell script.

This is where testing is more than just entering data and verifying the results. It takes a bit of extra work. Sometimes I add a command that simply prints the intermediate result of the code I just wrote and verify that. For more complex scripts, I add a -t option for "test mode." In this case, the internal test code executes only when the -t option is entered on the command line.

Final testing

After the code is complete, I go back to do a complete test of all the features and functions using known inputs to produce specific outputs. I also test some random inputs to see if the program can handle unexpected input.

Final testing is intended to verify that the program is functioning essentially as intended. A large part of the final test is to ensure that functions that worked earlier in the development cycle have not been broken by code that was added or changed later in the cycle.

If you have been testing the script as you add new code to it, you may think there should not be any surprises during the final test. Wrong! There are always surprises during final testing. Always. Expect those surprises, and be ready to spend time fixing them. If there were never any bugs discovered during final testing, there would be no point in doing a final test, would there?

Testing in production

Huh—what?

> "Not until a program has been in production for at least six months will the most harmful error be discovered."
> — Troutman's Programming Postulates

Yes, testing in production is now considered normal and desirable. Having been a tester myself, this seems reasonable. "But wait! That's dangerous," you say. My experience is that it is no more dangerous than extensive and rigorous testing in a dedicated test environment. In some cases, there is no choice because there is no test environment—only production.

Sysadmins are no strangers to the need to test new or revised scripts in production. Anytime a script is moved into production, that becomes the ultimate test. The production environment constitutes the most critical part of that test. Nothing that testers can dream up in a test environment can fully replicate the true production environment.

The allegedly new practice of testing in production is just the recognition of what sysadmins have known all along. The best test is production—so long as it is not the only test.

Fuzzy testing

This is another of those buzzwords that initially caused me to roll my eyes. Its essential meaning is simple: have someone bang on the keys until something happens, and see how well the program handles it. But there really is more to it than that.

Fuzzy testing is a bit like the time my son broke the code for a game in less than a minute with random input. That pretty much ended my attempts to write games for him.

Most test plans utilize very specific input that generates a specific result or output. Regardless of whether the test defines a positive or negative outcome as a success, it is still controlled, and the inputs and results are specified and expected, such as a specific error message for a specific failure mode.

Fuzzy testing is about dealing with randomness in all aspects of the test, such as starting conditions, very random and unexpected input, random combinations of options selected, low memory, high levels of CPU contending with other programs, multiple instances of the program under test, and any other random conditions that you can think of to apply to the tests.

I try to do some fuzzy testing from the beginning. If the Bash script cannot deal with significant randomness in its very early stages, then it is unlikely to get better as you add more code. This is a good time to catch these problems and fix them while the code is relatively simple. A bit of fuzzy testing at each stage is also useful in locating problems before they get masked by even more code.

After the code is completed, I like to do some more extensive fuzzy testing. Always do some fuzzy testing. I have certainly been surprised by some of the results. It is easy to test for the expected things, but users do not usually do the expected things with a script.

Previews of coming attractions

This part accomplished a little in the way of creating a template, but it mostly talked about testing. This is because testing is a critical part of creating any kind of program. In the next part in this guide, you will add a basic Help function along with some code to detect and act on options, such as -h, to your Bash script template.

Links

[1] https://opensource.com/article/18/7/how-be-lazy-sysadmin
[2] https://opensource.com/article/17/12/source-code-license

How to add a Help facility to your Bash program

In the third part in this guide, learn about using functions as you create a simple Help facility for your Bash script.

IN THE FIRST PART in this guide, you created a very small, one-line Bash script and explored the reasons for creating shell scripts and why they are the most efficient option for the system administrator, rather than compiled programs. In the second part, you began the task of creating a fairly simple template that you can use as a starting point for other Bash programs, then explored ways to test it.

This third of the four parts in this guide explains how to create and use a simple Help function. While creating your Help facility, you will also learn about using functions and how to handle command-line options such as -h.

Why Help?

Even fairly 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 so infrequently that I forget the exact syntax of the command I need. Others are so complex that I need to review the options and arguments even when I use them frequently.

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

About functions

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

Functions are called in scripts or from the command-line interface (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 execute when they are called, then the program flow sequence returns to the calling entity, and the next series of program statements in that entity executes.

The syntax of a function is:

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

Explore this by creating a simple function at the CLI. (The function is stored in the shell environment for the shell instance in which it is created.) You are 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" starter. Now, list all of the currently defined functions. There are a lot of them, so I am showing just the new **hw** function. When it is called from the command line or within a program, a function performs its programmed task and then exits and returns 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>
```

Remove that function because you do not need it anymore. You can do that with the **unset** command:

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

Creating the Help function

Open the **hello** program in an editor and add the Help function below to the **hello** program code after the copyright statement but before the echo **"Hello world!"** statement. This Help function will display a short description of the program, a syntax diagram, and short descriptions

of the available options. 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:

```
#################################################################
# 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 are typical for the programs I write, although none are in the code yet. Run the program to test it:

```
[student@testvm1 ~]$ ./hello
Add 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 have not added any logic to display Help only when you need it, the program will always display the Help. Since the function is working correctly, read on to add some logic to display the Help only when the -h option is used when you invoke the program at the command line.

Handling options

A Bash script's ability to handle command-line options such as **-h** gives some powerful capabilities to direct the program and modify what it does. In the case of the **-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 (see *How to program with Bash: Loops* [1] to learn more about **while**) 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. In the code below, the **while** command loops through the list of options by setting the variable **$options** for each. 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 evaluate the list of options until they have all been processed or it encounters an exit statement, 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 added to this case statement to delineate the end of each option.

Testing

Testing is now a little more complex. You need to test your program with a number of different options—and no options—to see how it responds. First, test with no options to ensure that it prints "Hello world!" as it should:

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

That works, so now test the logic that displays the Help text:

```
[student@testvm1 ~]$ ./hello -h
Add 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 try some testing to see what happens when you enter some unexpected options:

```
[student@testvm1 ~]$ ./hello -x
Hello world!
[student@testvm1 ~]$ ./hello -q
Hello world!
[student@testvm1 ~]$ ./hello -lkjsahdf
Add 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 ~]$
```

The program just ignores any options without specific responses without generating any errors. But notice the last entry (with **-lkjsahdf** for options): because there is an h in the list of options, the program recognizes it and prints the Help text. This testing has shown that the program doesn't have the ability to handle incorrect input and terminate the program if any is detected.

You can add another case stanza to the case statement to match any option that doesn't have an explicit match. This general case will match anything you have not provided a specific match for. The case statement now looks like this, with the catch-all match of **\?** as the last case. Any additional specific cases must precede this final one:

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

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

Where you are

You have accomplished a good amount in this part by adding the capability to process command-line options and a Help procedure. Your Bash script now looks like this:

```
#!/usr/bin/bash
############################################################
#                    scriptTemplate                        #
#                                                          #
# Use this template as the beginning of a new program. Place  #
# a short description of the script here.                  #
#                                                          #
# Change History                                           #
# 11/11/2019  David Both  Original code. This is a template   #
#                         for creating new Bash shell scripts.#
#                         Add new history entries as needed.  #
#                                                          #
#                                                          #
############################################################
############################################################
############################################################
#                                                          #
#  Copyright (C) 2007, 2019 David Both                     #
#  LinuxGeek46@both.org                                    #
#                                                          #
#  This program is free software; you can redistribute it     #
#  and/or modify ut under the terms of the GNU General Public #
#  License as published by the Free Software Foundation;      #
#  either version 2 of the License, or (at your option) any   #
#  later version.                                          #
#                                                          #
#  This program is distributed in the hope that it will be    #
#  useful, but WITHOUT ANY WARRANTY; without even the         #
#  implied warranty of MERCHANTABILITY or FITNESS FOR A       #
#  PARTICULAR PURPOSE.  See the GNU General Public License    #
#  for more details.                                       #
#                                                          #
#  You should have received a copy of the GNU General Public  #
#  License along with this program; if not, write to the Free #
#  Software Foundation, Inc., 59 Temple Place, Suite 330,     #
#  Boston, MA  02111-1307  USA                             #
#                                                          #
############################################################
############################################################
############################################################
############################################################
```

```
# Help                                                       #
############################################################
Help()
{
   # Display Help
   echo "Add description of the script functions here."
   echo
   echo "Syntax: scriptTemplate [-g|h|t|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;;
      \?) # incorrect option
         echo "Error: Invalid option"
         exit;;
   esac
done

echo "Hello world!"
```

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

Next time

In this part, you added a Help function as well as the ability to process command-line options to display it selectively. The program is getting a little more complex, so testing is becoming more important and requires more test paths in order to be complete.

The next part will look at initializing variables and doing a bit of sanity checking to ensure that the program will run under the correct set of conditions.

Links

[1] https://opensource.com/article/19/10/programming-bash-loops

Testing your Bash script

In the fourth and final part in this guide on automation with shell scripts, learn about initializing variables and ensuring your program runs correctly.

IN THE FIRST PART in this guide, you created your first, very small, one-line Bash script and explored the reasons for creating shell scripts. In the second part, you began creating a fairly simple template that can be a starting point for other Bash programs and began testing it. In the third part, you created and used a simple Help function and learned about using functions and how to handle command-line options such as **-h**.

This fourth and final part in the guide gets into variables and initializing them as well as how to do a bit of sanity testing to help ensure the program runs under the proper conditions. Remember, the objective of this guide is to build working code that will be used for a template for future Bash programming projects. The idea is to make getting started on new programming projects easy by having common elements already available in the template.

Variables

The Bash shell, like all programming languages, can deal with variables. A variable is a symbolic name that refers to a specific location in memory that contains a value of some sort. The value of a variable is changeable, i.e., it is variable. If you are not familiar with using variables, read my article *How to program with Bash: Syntax and tools* [1] before you go further.

Done? Great! Let's now look at some good practices when using variables.

I always set initial values for every variable used in my scripts. You can find this in your template script immediately after the procedures as the first part of the main program body, before it processes the options. Initializing each variable with an appropriate value can prevent errors that might occur with uninitialized variables in comparison or math operations. Placing this list of variables in one place allows you to see all of the variables that are supposed to be in the script and their initial values.

Your little script has only a single variable, **$option**, so far. Set it by inserting the following lines as shown:

```
############################################################
############################################################
# Main program                                             #
############################################################
############################################################
# Initialize variables
option=""
############################################################
# Process the input options. Add options as needed.        #
############################################################
```

Test this to ensure that everything works as it should and that nothing has broken as the result of this change.

Constants

Constants are variables, too—at least they should be. Use variables wherever possible in command-line interface (CLI) programs instead of hard-coded values. Even if you think you will use a particular value (such as a directory name, a file name, or a text string) just once, create a variable and use it where you would have placed the hard-coded name.

For example, the message printed as part of the main body of the program is a string literal, **echo "Hello world!"**. Change that to a variable. First, add the following statement to the variable initialization section:

```
Msg="Hello world!"
```

And now change the last line of the program from:

```
echo "Hello world!"
```

to:

```
echo "$Msg"
```

Test the results.

Sanity checks

Sanity checks are simply tests for conditions that need to be true in order for the program to work correctly, such as: the program must be run as the root user, or it must run on a particular distribution and release of that distro. Add a check for *root* as the running user in your simple program template.

Testing that the root user is running the program is easy because a program runs as the user that launches it.

The **id** command can be used to determine the numeric user ID (UID) the program is running under. It provides several bits of information when it is used without any options:

```
[student@testvm1 ~]$ id
uid=1001(student) gid=1001(student) groups=1001(student),5000(dev)
```

Using the **-u** option returns just the user's UID, which is easily usable in your Bash program:

```
[student@testvm1 ~]$ id -u
1001
[student@testvm1 ~]$
```

Add the following function to the program. I added it after the Help procedure, but you can place it anywhere in the procedures section. The logic is that if the UID is not zero, which is always the root user's UID, the program exits:

```
############################################################
# Check for root.                                          #
############################################################
CheckRoot()
{
   if [ `id -u` != 0 ]
   then
      echo "ERROR: You must be root user to run this program"
      exit
   fi
}
```

Now, add a call to the **CheckRoot** procedure just before the variable's initialization. Test this, first running the program as the student user:

```
[student@testvm1 ~]$ ./hello
ERROR: You must be root user to run this program
[student@testvm1 ~]$
```

then as the root user:

```
[root@testvm1 student]# ./hello
Hello world!
[root@testvm1 student]#
```

You may not always need this particular sanity test, so comment out the call to **CheckRoot** but leave all the code in place in the template. This way, all you need to do to use that code in a future program is to uncomment the call.

The code

After making the changes outlined above, your code should look like this:

```
#!/usr/bin/bash
############################################################
#                    scriptTemplate                        #
#                                                          #
# Use this template as the beginning of a new program. Place   #
# a short description of the script here.                  #
#                                                          #
# Change History                                           #
# 11/11/2019  David Both   Original code. This is a template   #
#                          for creating new Bash shell scripts.#
#                          Add new history entries as needed.  #
#                                                          #
#                                                          #
############################################################
############################################################
############################################################
#                                                          #
#  Copyright (C) 2007, 2019 David Both                     #
#  LinuxGeek46@both.org                                    #
#                                                          #
#  This program is free software; you can redistribute it      #
#  and/or modify it under the terms of the GNU General Public  #
#  License as published by the Free Software Foundation;       #
#  either version 2 of the License, or at your option) any     #
#  (later version.                                         #
#                                                          #
#  This program is distributed in the hope that it will be     #
#  useful, but WITHOUT ANY WARRANTY; without even the implied  #
#  warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR     #
#  PURPOSE.  See the GNU General Public License for more       #
#  details.                                                #
#                                                          #
#  You should have received a copy of the GNU General Public   #
#  License along with this program; if not, write to the Free #
#  Software Foundation, Inc., 59 Temple Place, Suite 330,      #
#  Boston, MA  02111-1307  USA                             #
#                                                          #
############################################################
############################################################
############################################################

############################################################
# 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
}

############################################################
# Check for root.                                          #
############################################################
CheckRoot()
{
   # If we are not running as root we exit the program
   if [ `id -u` != 0 ]
   then
      echo "ERROR: You must be root user to run this program"
      exit
   fi
}

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

############################################################
# Sanity checks                                            #
############################################################
# Are we rnning as root?
# CheckRoot

# Initialize variables
option=""
Msg="Hello world!"
############################################################
# Process the input options. Add options as needed.        #
############################################################
# Get the options
while getopts ":h" option; do
   case $option in
      h) # display Help
         Help
         exit;;
     \?) # incorrect option
         echo "Error: Invalid option"
         exit;;
   esac
done

echo "$Msg"
```

A final exercise

You probably noticed that the Help function in your code refers to features that are not in the code. As a final exercise, figure out how to add those functions to the code template you created.

Summary

In this part, you created a couple of functions to perform a sanity test for whether your program is running as root. Your program is getting a little more complex, so testing is becoming more important and requires more test paths to be complete.

This guide looked at a very minimal Bash program and how to build a script up a bit at a time. The result is a simple template that can be the starting point for other, more useful Bash scripts and that contains useful elements that make it easy to start new scripts.

By now, you get the idea: Compiled programs are necessary and fill a very important need. But for sysadmins, there is always a better way. Always use shell scripts to meet your job's automation needs. Shell scripts are open; their content and purpose are knowable. They can be readily modified to meet different requirements. I have never found anything that I need to do in my sysadmin role that cannot be accomplished with a shell script.

What you have created so far in this guide is just the beginning. As you write more Bash programs, you will find more bits of code that you use frequently and should be included in your program template.

Links

[1] https://opensource.com/article/19/10/programming-bash-syntax-tools