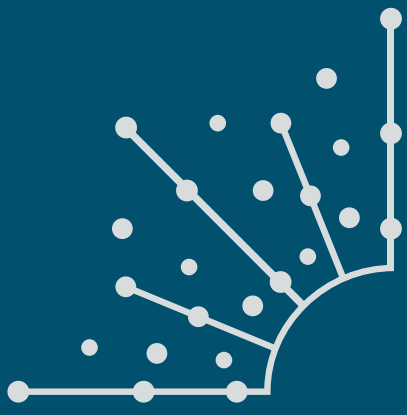
An introduction to programming with Bash



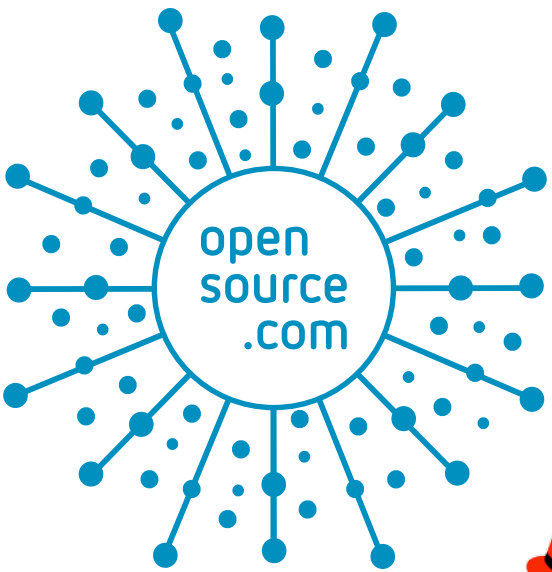
ABOUT OPENSOURCE.COM

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

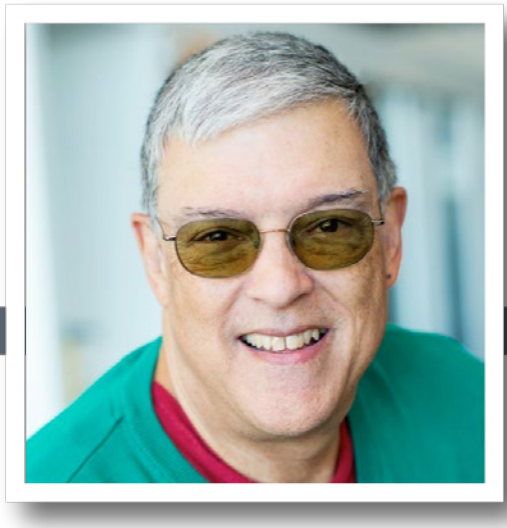
DAVID BOTH IS AN OPEN SOURCE Software and GNU/Linux advocate, trainer, writer, and speaker who lives in Raleigh North Carolina. He is a strong proponent of and evangelist for the "Linux Philosophy."

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

Syntax and tools

Learn basic Bash programming syntax and tools, as well as how to use variables and control operators, in the first part in this three-part guide.

A SHELL is the command interpreter for the operating system. Bash is my favorite shell, but every Linux shell interprets the commands typed by the user or sysadmin into a form the operating system can use. When the results are returned to the shell program, it sends them to STDOUT which, by default, displays them in the terminal [1]. All of the shells I am familiar with are also programming languages.

Features like tab completion, command-line recall and editing, and shortcuts like aliases all contribute to its value as a powerful shell. Its default command-line editing mode uses Emacs, but one of my favorite Bash features is that I can change it to Vi mode to use editing commands that are already part of my muscle memory.

However, if you think of Bash solely as a shell, you miss much of its true power. While researching my three-volume Linux self-study course [2] (on which this guide is based), I learned things about Bash that I'd never known in over 20 years of working with Linux. Some of these new bits of knowledge relate to its use as a programming language. Bash is a powerful programming language, one perfectly designed for use on the command line and in shell scripts.

This three-part guide explores using Bash as a command-line interface (CLI) programming language. This first part looks at some simple command-line programming with Bash, variables, and control operators. The other parts explore types of Bash files; string, numeric, and miscellaneous logical operators that provide execution-flow control logic; different types of shell expansions; and the **for**, **while**, and **until** loops that enable repetitive operations. They will also look at some commands that simplify and support the use of these tools.

The shell

A shell is the command interpreter for the operating system. Bash is my favorite shell, but every Linux shell interprets the commands typed by the user or sysadmin into a form the operating system can use. When the results are returned to the shell program, it displays them in the terminal. All of the shells I am familiar with are also programming languages.

Bash stands for Bourne Again Shell because the Bash shell is based upon [3] the older Bourne shell that was written by Steven Bourne in 1977. Many other shells [4] are available, but these are the four I encounter most frequently:

- **csh:** The C shell for programmers who like the syntax of the C language
- **ksh:** The Korn shell, written by David Korn and popular with Unix users
- **tcsh:** A version of csh with more ease-of-use features
- **zsh:** The Z shell, which combines many features of other popular shells

All shells have built-in commands that supplement or replace the ones provided by the core utilities. Open the shell's man page and find the "BUILT-INS" section to see the commands it provides.

Each shell has its own personality and syntax. Some will work better for you than others. I have used the C shell, the Korn shell, and the Z shell. I still like the Bash shell more than any of them. Use the one that works best for you, although that might require you to try some of the others. Fortunately, it's quite easy to change shells.

All of these shells are programming languages, as well as command interpreters. Here's a quick tour of some programming constructs and tools that are integral parts of Bash.

Bash as a programming language

Most sysadmins have used Bash to issue commands that are usually fairly simple and straightforward. But Bash can go beyond entering single commands, and many sysadmins create simple command-line programs to perform a series of tasks. These programs are common tools that can save time and effort.

My objective when writing CLI programs is to save time and effort (i.e., to be the lazy sysadmin). CLI programs support this by listing several commands in a specific sequence that execute one after another, so you do not need to watch the progress of one command and type in the next command when the first finishes. You can go do other things and not have to continually monitor the progress of each command.

What is "a program"?

The Free On-line Dictionary of Computing (FOLDOC) [5] defines a program as: "The instructions executed by a computer, as opposed to the physical device on which they run." Princeton University's WordNet [6] defines a program as: "…a sequence of instructions that a computer can interpret and execute…" Wikipedia [7] also has a good entry about computer programs.

Therefore, a program can consist of one or more instructions that perform a specific, related task. A computer program instruction is also called a program statement. For sysadmins, a program is usually a sequence of shell commands. All the shells available for Linux, at least the ones I am familiar with, have at least a basic form of programming capability, and Bash, the default shell for most Linux distributions, is no exception.

While this guide uses Bash (because it is so ubiquitous), if you use a different shell, the general programming concepts will be the same, although the constructs and syntax may differ somewhat. Some shells may support some features that others do not, but they all provide some programming capability. Shell programs can be stored in a file for repeated use, or they may be created on the command line as needed.

Simple CLI programs

The simplest command-line programs are one or two consecutive program statements, which may be related or not, that are entered on the command line before the **Enter** key is pressed. The second statement in a program, if there is one, might be dependent upon the actions of the first, but it does not need to be.

There is also one bit of syntactical punctuation that needs to be clearly stated. When entering a single command on the command line, pressing the **Enter** key terminates the command with an implicit semicolon (**;**). When used in a CLI shell program entered as a single line on the command line, the semicolon must be used to terminate each statement and separate it from the next one. The last statement in a CLI shell program can use an explicit or implicit semicolon.

Some basic syntax

The following examples will clarify this syntax. This program consists of a single command with an explicit terminator:

```
[student@studentvm1 ~]$ echo "Hello world." ;
Hello world.
```

That may not seem like much of a program, but it is the first program I encounter with every new programming language I learn. The syntax may be a bit different for each language, but the result is the same.

Let's expand a little on this trivial but ubiquitous program. Your results will be different from mine because I have done other experiments, while you may have only the default directories and files that are created in the account home directory the first time you log into an account via the GUI desktop.

```
[student@studentvm1 ~]$ echo "My home directory." ; ls ;
My home directory.
chapter25   TestFile1.Linux  dmesg2.txt  Downloads  newfile.txt
  softlink1  testdir6
chapter26   TestFile1.mac    dmesg3.txt  file005    Pictures
  Templates  testdir
TestFile1      Desktop       dmesg.txt   link3      Public
  testdir    Videos
TestFile1.dos  dmesg1.txt    Documents   Music      random.txt
  testdir1
```

That makes a bit more sense. The results are related, but the individual program statements are independent of each other. Notice that I like to put spaces before and after the semicolon because it makes the code a bit easier to read. Try that little CLI program again without an explicit semicolon at the end:

```
[student@studentvm1 ~]$ echo "My home directory." ; ls
```

There is no difference in the output.

Something about variables

Like all programming languages, the Bash shell 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.

Bash does not type variables like C and related languages, defining them as integers, floating points, or string types. In Bash, all variables are strings. A string that is an integer can be used in integer arithmetic, which is the only type of math that Bash is capable of doing. If more complex math is required, the **bc** command can be used in CLI programs and scripts.

Variables are assigned values and can be used to refer to those values in CLI programs and scripts. The value of a variable is set using its name but not preceded by a **$** sign. The assignment **VAR=10** sets the value of the variable VAR to 10. To print the value of the variable, you can use the statement **echo $VAR**. Start with text (i.e., non-numeric) variables.

Bash variables become part of the shell environment until they are unset.

Check the initial value of a variable that has not been assigned; it should be null. Then assign a value to the variable and print it to verify its value. You can do all of this in a single CLI program:

```
[student@studentvm1 ~]$ echo $MyVar ; MyVar="Hello World" ;
  echo $MyVar ;

Hello World
[student@studentvm1 ~]$
```

Note: The syntax of variable assignment is very strict. There must be no spaces on either side of the equal (=) sign in the assignment statement.

The empty line indicates that the initial value of **MyVar** is null. Changing and setting the value of a variable are done the same way. This example shows both the original and the new value.

As mentioned, Bash can perform integer arithmetic calculations, which is useful for calculating a reference to the location of an element in an array or doing simple math problems. It is not suitable for scientific computing or anything that requires decimals, such as financial calculations. There are much better tools for those types of calculations.

Here's a simple calculation:

```
[student@studentvm1 ~]$ Var1="7" ; Var2="9" ; echo "Result =
  $((Var1*Var2))"
Result = 63
```

What happens when you perform a math operation that results in a floating-point number?

```
[student@studentvm1 ~]$ Var1="7" ; Var2="9" ; echo "Result =
  $((Var1/Var2))"
Result = 0
[student@studentvm1 ~]$ Var1="7" ; Var2="9" ; echo "Result =
  $((Var2/Var1))"
Result = 1
[student@studentvm1 ~]$
```

The result is the nearest integer. Notice that the calculation was performed as part of the **echo** statement. The math is performed before the enclosing echo command due to the Bash order of precedence. For details see the Bash man page and search "precedence."

Control operators

Shell control operators are one of the syntactical operators for easily creating some interesting command-line programs. The simplest form of CLI program is just stringing several commands together in a sequence on the command line:

```
command1 ; command2 ; command3 ; command4 ; . . . ; etc. ;
```

Those commands all run without a problem so long as no errors occur. But what happens when an error occurs? You can anticipate and allow for errors using the built-in **&&** and **||** Bash control operators. These two control operators provide some flow control and enable you to alter the sequence of code execution. The semicolon is also considered to be a Bash control operator, as is the newline character.

The **&&** operator simply says, "if command1 is successful, then run command2. If command1 fails for any reason, then command2 is skipped." That syntax looks like this:

```
command1 && command2
```

Now, look at some commands that will create a new directory and—if it's successful—make it the present working directory (PWD). Ensure that your home directory (**~**) is the PWD. Try this first in **/root**, a directory that you do not have access to:

```
[student@studentvm1 ~]$ Dir=/root/testdir ; mkdir $Dir/ && cd $Dir
mkdir: cannot create directory '/root/testdir/': Permission denied
[student@studentvm1 ~]$
```

The error was emitted by the **mkdir** command. You did not receive an error indicating that the file could not be created because the creation of the directory failed. The **&&** control operator sensed the non-zero return code, so the **touch** command was skipped. Using the **&&** control operator prevents the **touch** command from running because there was an error in creating the directory. This type of command-line program flow control can prevent errors from compounding and making a real mess of things. But it's time to get a little more complicated.

The **||** control operator allows you to add another program statement that executes when the initial program statement returns a code greater than zero. The basic syntax looks like this:

```
command1 || command2
```

This syntax reads, "If command1 fails, execute command2." That implies that if command1 succeeds, command2 is skipped. Try this by attempting to create a new directory:

```
[student@studentvm1 ~]$ Dir=/root/testdir ; mkdir $Dir || echo
  "$Dir was not created."
mkdir: cannot create directory '/root/testdir': Permission
  denied
/root/testdir was not created.
[student@studentvm1 ~]$
```

This is exactly what you would expect. Because the new directory could not be created, the first command failed, which resulted in the execution of the second command.

Combining these two operators provides the best of both. The control operator syntax using some flow control takes this general form when the **&&** and **||** control operators are used:

```
preceding commands ; command1 && command2 || command3 ;
  following commands
```

This syntax can be stated like so: "If command1 exits with a return code of 0, then execute command2, otherwise execute command3." Try it:

```
[student@studentvm1 ~]$ Dir=/root/testdir ; mkdir $Dir && cd
  $Dir || echo "$Dir was not created."
mkdir: cannot create directory '/root/testdir': Permission
  denied
/root/testdir was not created.
[student@studentvm1 ~]$
```

Now try the last command again using your home directory instead of the **/root** directory. You will have permission to create this directory:

```
[student@studentvm1 ~]$ Dir=~/testdir ; mkdir $Dir && cd $Dir
  || echo "$Dir was not created."
[student@studentvm1 testdir]$
```

The control operator syntax, like **command1 && command2**, works because every command sends a return code (RC) to the shell that indicates if it completed successfully or whether there was some type of failure during execution. By convention, an RC of zero (0) indicates success, and any positive number indicates some type of failure. Some of the tools sysadmins use just return a one (1) to indicate a failure, but many use other codes to indicate the type of failure that occurred.

The Bash shell variable **$?** contains the RC from the last command. This RC can be checked very easily by a script, the next command in a list of commands, or even the sysadmin directly. Start by running a simple command and immediately checking the RC. The RC will always be for the last command that ran before you looked at it.

```
[student@studentvm1 testdir]$ ll ; echo "RC = $?"
total 1264
drwxrwxr-x  2 student student   4096 Mar  2 08:21 chapter25
drwxrwxr-x  2 student student   4096 Mar 21 15:27 chapter26
-rwxr-xr-x  1 student student     92 Mar 20 15:53 TestFile1
<snip>
drwxrwxr-x. 2 student student 663552 Feb 21 14:12 testdir
drwxr-xr-x. 2 student student   4096 Dec 22 13:15 Videos
RC = 0
[student@studentvm1 testdir]$
```

The RC, in this case, is zero, which means the command completed successfully. Now try the same command on root's home directory, a directory you do not have permissions for:

```
[student@studentvm1 testdir]$ ll /root ; echo "RC = $?"
ls: cannot open directory '/root': Permission denied
RC = 2
[student@studentvm1 testdir]$
```

In this case, the RC is two; this means permission was denied for a non-root user to access a directory to which the user is not permitted access. The control operators use these RCs to enable you to alter the sequence of program execution.

Summary

This part looked at Bash as a programming language and explored its basic syntax as well as some basic tools. It showed how to print data to STDOUT and how to use variables and control operators. The next part in this guide looks at some of the many Bash logical operators that control the flow of instruction execution.

Links

[1] https://opensource.com/article/18/10/linux-data-streams
[2] http://www.both.org/?page_id=1183
[3] https://opensource.com/19/9/command-line-heroes-bash
[4] https://en.wikipedia.org/wiki/Comparison_of_command_shells
[5] http://foldoc.org/program
[6] https://wordnet.princeton.edu/
[7] https://en.wikipedia.org/wiki/Computer_program

Logical operators and shell expansions

Learn about logical operators and shell expansions, in the second part in this three-part guide on programming with Bash.

LOGICAL OPERATORS are the basis for making decisions in a program and executing different sets of instructions based on those decisions. This is sometimes called flow control.

Logical operators

Bash has a large set of logical operators that can be used in conditional expressions. The most basic form of the **if** control structure tests for a condition and then executes a list of program statements if the condition is true. There are three types of operators: file, numeric, and non-numeric operators. Each operator returns true (0) if the condition is met and false (1) if the condition is not met.

The functional syntax of these comparison operators is one or two arguments with an operator that are placed within square braces, followed by a list of program statements that are executed if the condition is true, and an optional list of program statements if the condition is false:

```
if [ arg1 operator arg2 ] ; then list
or
if [ arg1 operator arg2 ] ; then list ; else list ; fi
```

The spaces in the comparison are required as shown. The single square braces, **[** and **]**, are the traditional Bash symbols that are equivalent to the **test** command:

```
if test arg1 operator arg2 ; then list
```

There is also a more recent syntax that offers a few advantages and that some sysadmins prefer. This format is a bit less compatible with different versions of Bash and other shells, such as ksh (the Korn shell). It looks like:

```
if [[ arg1 operator arg2 ]] ; then list
```

File operators

File operators are a powerful set of logical operators within Bash. Figure 1 lists more than 20 different operators that Bash can perform on files. I use them quite frequently in my scripts.

Operator	Description
-a filename	True if the file exists; it can be empty or have some content but, so long as it exists, this will be true
-b filename	True if the file exists and is a block special file such as a hard drive like **/dev/sda** or **/dev/sda1**
-c filename	True if the file exists and is a character special file such as a TTY device like **/dev/TTY1**
-d filename	True if the file exists and is a directory
-e filename	True if the file exists; this is the same as **-a** above
-f filename	True if the file exists and is a regular file, as opposed to a directory, a device special file, or a link, among others
-g filename	True if the file exists and is **set-group-id**, **SETGID**
-h filename	True if the file exists and is a symbolic link
-k filename	True if the file exists and its “sticky” bit is set
-p filename	True if the file exists and is a named pipe (FIFO)
-r filename	True if the file exists and is readable, i.e., has its read bit set
-s filename	True if the file exists and has a size greater than zero; a file that exists but that has a size of zero will return false
-t fd	True if the file descriptor **fd** is open and refers to a terminal
-u filename	True if the file exists and its **set-user-id** bit is set
-w filename	True if the file exists and is writable
-x filename	True if the file exists and is executable

-G filename	True if the file exists and is owned by the effective group ID
-L filename	True if the file exists and is a symbolic link
-N filename	True if the file exists and has been modified since it was last read
-O filename	True if the file exists and is owned by the effective user ID
-S filename	True if the file exists and is a socket
file1 -ef file2	True if file1 and file2 refer to the same device and iNode numbers
file1 -nt file2	True if file1 is newer (according to modification date) than file2, or if file1 exists and file2 does not
file1 -ot file2	True if file1 is older than file2, or if file2 exists and file1 does not

Fig. 1: The Bash file operators

As an example, start by testing for the existence of a file:

```
[student@studentvm1 testdir]$ File="TestFile1" ; if [ -e $File ]
  ; then echo "The file $File exists." ; else echo "The file $File
  does not exist." ; fi
The file TestFile1 does not exist.
[student@studentvm1 testdir]$
```

Next, create a file for testing named **TestFile1**. For now, it does not need to contain any data:

```
[student@studentvm1 testdir]$ touch TestFile1
```

It is easy to change the value of the **$File** variable rather than a text string for the file name in multiple locations in this short CLI program:

```
[student@studentvm1 testdir]$ File="TestFile1" ; if [ -e $File ]
  ; then echo "The file $File exists." ; else echo "The file $File
  does not exist." ; fi
The file TestFile1 exists.
[student@studentvm1 testdir]$
```

Now, run a test to determine whether a file exists and has a non-zero length, which means it contains data. You want to test for three conditions: 1. the file does not exist; 2. the file exists and is empty; and 3. the file exists and contains data. Therefore, you need a more complex set of tests—use the **elif** stanza in the **if-elif-else** construct to test for all of the conditions:

```
[student@studentvm1 testdir]$ File="TestFile1" ; if [ -s $File ]
  ; then echo "$File exists and contains data." ; fi
[student@studentvm1 testdir]$
```

In this case, the file exists but does not contain any data. Add some data and try again:

```
[student@studentvm1 testdir]$ File="TestFile1" ; echo "This
  is file $File" > $File ; if [ -s $File ] ; then echo "$File
  exists and contains data." ; fi
TestFile1 exists and contains data.
[student@studentvm1 testdir]$
```

That works, but it is only truly accurate for one specific condition out of the three possible ones. Add an **else** stanza so you can be somewhat more accurate, and delete the file so you can fully test this new code:

```
[student@studentvm1 testdir]$ File="TestFile1" ; rm $File ; if
  [ -s $File ] ; then echo "$File exists and contains data." ;
  else echo "$File does not exist or is empty." ; fi
TestFile1 does not exist or is empty.
```

Now create an empty file to test:

```
[student@studentvm1 testdir]$ File="TestFile1" ; touch $File ;
  if [ -s $File ] ; then echo "$File exists and contains data." ;
  else echo "$File does not exist or is empty." ; fi
TestFile1 does not exist or is empty.
```

Add some content to the file and test again:

```
[student@studentvm1 testdir]$ File="TestFile1" ; echo "This
  is file $File" > $File ; if [ -s $File ] ; then echo "$File
  exists and contains data." ; else echo "$File does not exist
  or is empty." ; fi
TestFile1 exists and contains data.
```

Now, add the **elif** stanza to discriminate between a file that does not exist and one that is empty:

```
[student@studentvm1 testdir]$ File="TestFile1" ; touch $File ;
  if [ -s $File ] ; then echo "$File exists and contains data." ;
  elif [ -e $File ] ; then echo "$File exists and is empty." ;
  else echo "$File does not exist." ; fi
TestFile1 exists and is empty.
[student@studentvm1 testdir]$ File="TestFile1" ; echo "This is
  $File" > $File ; if [ -s $File ] ; then echo "$File exists and
  contains data." ; elif [ -e $File ] ; then echo "$File exists
  and is empty." ; else echo "$File does not exist." ; fi
TestFile1 exists and contains data.
[student@studentvm1 testdir]$
```

Now you have a Bash CLI program that can test for these three different conditions… but the possibilities are endless.

It is easier to see the logic structure of the more complex compound commands if you arrange the program statements more like you would in a script that you can save in a file. Figure 2 shows how this would look. The indents of the program statements in each stanza of the **if-elif-else** structure help to clarify the logic.

```
File="TestFile1"
echo "This is $File" > $File
if [ -s $File ]
   then
   echo "$File exists and contains data."
elif [ -e $File ]
   then
   echo "$File exists and is empty."
else
   echo "$File does not exist."
fi
```

Fig. 2: The command line program rewritten as it would appear in a script

Logic this complex is too lengthy for most CLI programs. Although any Linux or Bash built-in commands may be used in CLI programs, as the CLI programs get longer and more complex, it makes more sense to create a script that is stored in a file and can be executed at any time, now or in the future.

String comparison operators

String comparison operators enable the comparison of alphanumeric strings of characters. There are only a few of these operators, which are listed in Figure 3.

Operator	Description
-z string	True if the length of string is zero
-n string	True if the length of string is non-zero
string1 == string2 or string1 = string2	True if the strings are equal; a single = should be used with the test command for POSIX conformance. When used with the **[[** command, this performs pattern matching as described above (compound commands).
string1 != string2	True if the strings are not equal
string1 < string2	True if string1 sorts before string2 lexicographically (refers to locale-specific sorting sequences for all alphanumeric and special characters)
string1 > string2	True if string1 sorts after string2 lexicographically

Fig. 3: Bash string logical operators

First, look at string length. The quotes around **$MyVar** in the comparison must be there for the comparison to work. (You should still be working in **~/testdir**.)

```
[student@studentvm1 testdir]$ MyVar="" ; if [ -z "" ] ; then
  echo "MyVar is zero length." ; else echo "MyVar contains
  data" ; fi
MyVar is zero length.
```

```
[student@studentvm1 testdir]$ MyVar="Random text" ; if [ -z
  "" ] ; then echo "MyVar is zero length." ; else echo "MyVar
  contains data" ; fi
MyVar is zero length.
```

You could also do it this way:

```
[student@studentvm1 testdir]$ MyVar="Random text" ; if [ -n
  "$MyVar" ] ; then echo "MyVar contains data." ; else echo
  "MyVar is zero length" ; fi
MyVar contains data.
[student@studentvm1 testdir]$ MyVar="" ; if [ -n "$MyVar" ] ;
  then echo "MyVar contains data." ; else echo "MyVar is zero
  length" ; fi
MyVar is zero length
```

Sometimes you may need to know a string's exact length. This is not a comparison, but it is related. Unfortunately, there is no simple way to determine the length of a string. There are a couple of ways to do it, but I think using the **expr** (evaluate expression) command is easiest. Read the man page for **expr** for more about what it can do. Note that quotes are required around the string or variable you're testing.

```
[student@studentvm1 testdir]$ MyVar="" ; expr length "$MyVar"
0
[student@studentvm1 testdir]$ MyVar="How long is this?" ; expr
  length "$MyVar"
17
[student@studentvm1 testdir]$ expr length "We can also find the
  length of a literal string as well as a variable."
70
```

Regarding comparison operators, I use a lot of testing in my scripts to determine whether two strings are equal (i.e., identical). I use the non-POSIX version of this comparison operator:

```
[student@studentvm1 testdir]$ Var1="Hello World" ; Var2="Hello
  World" ; if [ "$Var1" == "$Var2" ] ; then echo "Var1 matches
  Var2" ; else echo "Var1 and Var2 do not match." ; fi
Var1 matches Var2
[student@studentvm1 testdir]$ Var1="Hello World" ; Var2="Hello
  world" ; if [ "$Var1" == "$Var2" ] ; then echo "Var1 matches
  Var2" ; else echo "Var1 and Var2 do not match." ; fi
Var1 and Var2 do not match.
```

Experiment some more on your own to try out these operators.

Numeric comparison operators

Numeric operators make comparisons between two numeric arguments. Like the other operator classes, most are easy to understand.

Operator	Description
arg1 -eq arg2	True if arg1 equals arg2
arg1 -ne arg2	True if arg1 is not equal to arg2
arg1 -lt arg2	True if arg1 is less than arg2
arg1 -le arg2	True if arg1 is less than or equal to arg2
arg1 -gt arg2	True if arg1 is greater than arg2
arg1 -ge arg2	True if arg1 is greater than or equal to arg2

Fig. 4: Bash numeric comparison logical operators

Here are some simple examples. The first instance sets the variable **$X** to 1, then tests to see if **$X** is equal to 1. In the second instance, **X** is set to 0, so the comparison is not true.

```
[student@studentvm1 testdir]$ X=1 ; if [ $X -eq 1 ] ; then echo
  "X equals 1" ; else echo "X does not equal 1" ; fi
X equals 1
[student@studentvm1 testdir]$ X=0 ; if [ $X -eq 1 ] ; then echo
  "X equals 1" ; else echo "X does not equal 1" ; fi
X does not equal 1
[student@studentvm1 testdir]$
```

Try some more experiments on your own.

Miscellaneous operators

These miscellaneous operators show whether a shell option is set or a shell variable has a value, but it does not discover the value of the variable, just whether it has one.

Operator	Description
-o optname	True if the shell option optname is enabled (see the list of options under the description of the -o option to the Bash set builtin in the Bash man page)
-v varname	True if the shell variable varname is set (has been assigned a value)
-R varname	True if the shell variable varname is set and is a name reference

Fig. 5: Miscellaneous Bash logical operators

Experiment on your own to try out these operators.

Expansions

Bash supports a number of types of expansions and substitutions that can be quite useful. According to the Bash man page, Bash has seven forms of expansions. This part looks at five of them: tilde expansion, arithmetic expansion, pathname expansion, brace expansion, and command substitution.

Brace expansion

Brace expansion is a method for generating arbitrary strings. (This tool is used below to create a large number of files for experiments with special pattern characters.) Brace expansion can be used to generate lists of arbitrary strings and insert them into a specific location within an enclosing static string or at either end of a static string. This may be hard to visualize, so it's best to just do it.

First, here's what a brace expansion does:

```
[student@studentvm1 testdir]$ echo {string1,string2,string3}
string1 string2 string3
```

Well, that is not very helpful, is it? But look what happens when you use it just a bit differently:

```
[student@studentvm1 testdir]$ echo "Hello "{David,Jen,Rikki,Jason}.
Hello David. Hello Jen. Hello Rikki. Hello Jason.
```

That looks like something useful—it could save a good deal of typing. Now try this:

```
[student@studentvm1 testdir]$ echo b{ed,olt,ar}s
beds bolts bars
```

I could go on, but you get the idea.

Tilde expansion

Arguably, the most common expansion is the tilde (~) expansion. When you use this in a command like **cd ~/Documents**, the Bash shell expands it as a shortcut to the user's full home directory.

Use these Bash programs to observe the effects of the tilde expansion:

```
[student@studentvm1 testdir]$ echo ~
/home/student
[student@studentvm1 testdir]$ echo ~/Documents
/home/student/Documents
[student@studentvm1 testdir]$ Var1=~/Documents ; echo $Var1 ;
  cd $Var1
/home/student/Documents
[student@studentvm1 Documents]$
```

Pathname expansion

Pathname expansion is a fancy term expanding file-globbing patterns, using the characters **?** and *****, into the full names of directories that match the pattern. File globbing refers to special pattern characters that enable significant flexibility in matching file names, directories, and other strings when performing various actions. These special pattern characters allow matching single, multiple, or specific characters in a string.

- **?** — Matches only one of any character in the specified location within the string
- ***** — Matches zero or more of any character in the specified location within the string

This expansion is applied to matching directory names. To see how this works, ensure that **testdir** is the present working directory (PWD) and start with a plain listing (the contents of my home directory will be different from yours):

```
[student@studentvm1 testdir]$ ls
chapter6  cpuHog.dos    dmesg1.txt  Documents  Music
  softlink1  testdir6    Videos
chapter7  cpuHog.Linux  dmesg2.txt  Downloads  Pictures
  Templates  testdir
testdir  cpuHog.mac    dmesg3.txt  file005    Public
  testdir    tmp
cpuHog     Desktop       dmesg.txt   link3      random.txt
  testdir1   umask.test
[student@studentvm1 testdir]$
```

Now list the directories that start with **Do**, **testdir/Documents**, and **testdir/Downloads**:

```
Documents:
Directory01  file07  file15       test02  test10  test20
  testfile13  TextFiles
Directory02  file08  file16       test03  test11  testfile01
  testfile14
file01       file09  file17       test04  test12  testfile04
  testfile15
file02       file10  file18       test05  test13  testfile05
  testfile16
file03       file11  file19       test06  test14  testfile09
  testfile17
file04       file12  file20       test07  test15  testfile10
  testfile18
file05       file13  Student1.txt test08  test16  testfile11
  testfile19
file06       file14  test01       test09  test18  testfile12
testfile20

Downloads:
[student@studentvm1 testdir]$
```

Well, that did not do what you wanted. It listed the contents of the directories that begin with **Do**. To list only the directories and not their contents, use the **-d** option.

```
[student@studentvm1 testdir]$ ls -d Do*
Documents  Downloads
[student@studentvm1 testdir]$
```

In both cases, the Bash shell expands the **Do*** pattern into the names of the two directories that match the pattern. But what if there are also files that match the pattern?

```
[student@studentvm1 testdir]$ touch Downtown ; ls -d Do*
Documents  Downloads  Downtown
[student@studentvm1 testdir]$
```

This shows the file, too. So any files that match the pattern are also expanded to their full names.

Command substitution

Command substitution is a form of expansion that allows the STDOUT data stream of one command to be used as the argument of another command; for example, as a list of items to be processed in a loop. The Bash man page says: "Command substitution allows the output of a command to replace the command name." I find that to be accurate if a bit obtuse.

There are two forms of this substitution, **`command`** and **$(command)**. In the older form using back tics (`), using a backslash (\) in the command retains its literal meaning. However, when it's used in the newer parenthetical form, the backslash takes on its meaning as a special character. Note also that the parenthetical form uses only single parentheses to open and close the command statement.

I frequently use this capability in command-line programs and scripts where the results of one command can be used as an argument for another command.

Start with a very simple example that uses both forms of this expansion (again, ensure that **testdir** is the PWD):

```
[student@studentvm1 testdir]$ echo "Todays date is `date`"
Todays date is Sun Apr  7 14:42:46 EDT 2019
[student@studentvm1 testdir]$ echo "Todays date is $(date)"
Todays date is Sun Apr  7 14:42:59 EDT 2019
[student@studentvm1 testdir]$
```

The **-w** option to the **seq** utility adds leading zeros to the numbers generated so that they are all the same width, i.e., the same number of digits regardless of the value. This makes it easier to sort them in numeric sequence.

The **seq** utility is used to generate a sequence of numbers:

```
[student@studentvm1 testdir]$ seq 5
1
2
3
4
5
[student@studentvm1 testdir]$ echo `seq 5`
1 2 3 4 5
[student@studentvm1 testdir]$
```

Now you can do something a bit more useful, like creating a large number of empty files for testing:

```
[student@studentvm1 testdir]$ for I in $(seq -w 5000) ; do
  touch file-$I ; done
```

In this usage, the statement **seq -w 5000** generates a list of numbers from one to 5,000. By using command substitution as part of the **for** statement, the list of numbers is used by the **for** statement to generate the numerical part of the file names.

Arithmetic expansion

Bash can perform integer math, but it is rather cumbersome (as you will soon see). The syntax for arithmetic expansion is **$((arithmetic-expression))**, using double parentheses to open and close the expression.

Arithmetic expansion works like command substitution in a shell program or script; the value calculated from the expression replaces the expression for further evaluation by the shell.

Once again, start with something simple:

```
[student@studentvm1 testdir]$ echo $((1+1))
2
[student@studentvm1 testdir]$ Var1=5 ; Var2=7 ;
  Var3=$((Var1*Var2)) ; echo "Var 3 = $Var3"
Var 3 = 35
```

The following division results in zero because the result would be a decimal value of less than one:

```
[student@studentvm1 testdir]$ Var1=5 ; Var2=7 ; Var3=$((Var1/
  Var2)) ; echo "Var 3 = $Var3"
Var 3 = 0
```

Here is a simple calculation I often do in a script or CLI program that tells me how much total virtual memory I have in a Linux host. The **free** command does not provide that data:

```
[student@studentvm1 testdir]$ RAM=`free | grep ^Mem | awk
  '{print $2}'` ; Swap=`free | grep ^Swap | awk '{print $2}'`
  ; echo "RAM = $RAM and Swap = $Swap" ; echo "Total Virtual
  memory is $((RAM+Swap))" ;
RAM = 4037080 and Swap = 6291452
Total Virtual memory is 10328532
```

I used the ` character to delimit the sections of code used for command substitution.

I use Bash arithmetic expansion mostly for checking system resource amounts in a script and then choose a program execution path based on the result.

Summary

This part, the second in this guide on Bash as a programming language, explored the Bash file, string, numeric, and miscellaneous logical operators that provide execution-flow control logic and the different types of shell expansions.

The third part in this guide will explore the use of loops for performing various types of iterative operations.

Loops

Learn how to use loops for performing iterative operations, in the final part in this three-part guide on programming with Bash.

EVERY PROGRAMMING LANGUAGE I have ever used has at least a couple types of loop structures that provide various capabilities to perform repetitive operations. I use the for loop quite often but I also find the while and until loops useful.

for loops

Bash's implementation of the **for** command is, in my opinion, a bit more flexible than most because it can handle non-numeric values; in contrast, for example, the standard C language **for** loop can deal only with numeric values.

The basic structure of the Bash version of the **for** command is simple:

```
for Var in list1 ; do list2 ; done
```

This translates to: "For each value in list1, set the **$Var** to that value and then perform the program statements in list2 using that value; when all of the values in list1 have been used, it is finished, so exit the loop." The values in list1 can be a simple, explicit string of values, or they can be the result of a command substitution (described in the second part in the guide). I use this construct frequently.

To try it, ensure that **~/testdir** is still the present working directory (PWD). Clean up the directory, then look at a trivial example of the **for** loop starting with an explicit list of values. This list is a mix of alphanumeric values—but do not forget that all variables are strings and can be treated as such.

```
[student@studentvm1 testdir]$ rm *
[student@studentvm1 testdir]$ for I in a b c d 1 2 3 4 ; do
   echo $I ; done
a
b
c
d
1
2
3
4
```

Here is a bit more useful version with a more meaningful variable name:

```
[student@studentvm1 testdir]$ for Dept in "Human Resources"
   Sales Finance "Information Technology" Engineering
   Administration Research ; do echo "Department $Dept" ; done
Department Human Resources
Department Sales
Department Finance
Department Information Technology
Department Engineering
Department Administration
Department Research
```

Make some directories (and show some progress information while doing so):

```
[student@studentvm1 testdir]$ for Dept in "Human Resources"
  Sales Finance "Information Technology" Engineering
  Administration Research ; do echo "Working on Department
  $Dept" ; mkdir "$Dept"  ; done
Working on Department Human Resources
Working on Department Sales
Working on Department Finance
Working on Department Information Technology
Working on Department Engineering
Working on Department Administration
Working on Department Research
[student@studentvm1 testdir]$ ll
total 28
drwxrwxr-x 2 student student 4096 Apr  8 15:45  Administration
drwxrwxr-x 2 student student 4096 Apr  8 15:45  Engineering
drwxrwxr-x 2 student student 4096 Apr  8 15:45  Finance
drwxrwxr-x 2 student student 4096 Apr  8 15:45 'Human Resources'
drwxrwxr-x 2 student student 4096 Apr  8 15:45 'Information
  Technology'
drwxrwxr-x 2 student student 4096 Apr  8 15:45  Research
drwxrwxr-x 2 student student 4096 Apr  8 15:45  Sales
```

The **$Dept** variable must be enclosed in quotes in the **mkdir** statement; otherwise, two-part department names (such as "Information Technology") will be treated as two separate departments. That highlights a best practice I like to follow: all file and directory names should be a single word. Although most modern operating systems can deal with spaces in names, it takes extra work for sysadmins to ensure that

those special cases are considered in scripts and CLI programs. (They almost certainly should be considered, even if they're annoying because you never know what files you will have.)

So, delete everything in **~/testdir**—again—and do this one more time:

```
[student@studentvm1 testdir]$ rm -rf * ; ll
total 0
[student@studentvm1 testdir]$ for Dept in Human-Resources Sales
Finance Information-Technology Engineering Administration
Research ; do echo "Working on Department $Dept" ; mkdir
"$Dept"  ; done
Working on Department Human-Resources
Working on Department Sales
Working on Department Finance
Working on Department Information-Technology
Working on Department Engineering
Working on Department Administration
Working on Department Research
[student@studentvm1 testdir]$ ll
total 28
drwxrwxr-x 2 student student 4096 Apr  8 15:52 Administration
drwxrwxr-x 2 student student 4096 Apr  8 15:52 Engineering
drwxrwxr-x 2 student student 4096 Apr  8 15:52 Finance
drwxrwxr-x 2 student student 4096 Apr  8 15:52 Human-Resources
drwxrwxr-x 2 student student 4096 Apr  8 15:52 Information-
  Technology
drwxrwxr-x 2 student student 4096 Apr  8 15:52 Research
drwxrwxr-x 2 student student 4096 Apr  8 15:52 Sales
```

Suppose someone asks for a list of all RPMs on a particular Linux computer and a short description of each. This happened to me when I worked for the State of North Carolina. Since open source was not "approved" for use by state agencies at that time, and I only used Linux on my desktop computer, the pointy-haired bosses (PHBs) needed a list of each piece of software that was installed on my computer so that they could "approve" an exception.

How would you approach that? Here is one way, starting with the knowledge that the **rpm –qa** command provides a complete description of an RPM, including the two items the PHBs want: the software name and a brief summary.

Build up to the final result one step at a time. First, list all RPMs:

```
[student@studentvm1 testdir]$ rpm -qa
perl-HTTP-Message-6.18-3.fc29.noarch
perl-IO-1.39-427.fc29.x86_64
perl-Math-Complex-1.59-429.fc29.noarch
lua-5.3.5-2.fc29.x86_64
java-11-openjdk-headless-11.0.ea.28-2.fc29.x86_64
util-linux-2.32.1-1.fc29.x86_64
libreport-fedora-2.9.7-1.fc29.x86_64
rpcbind-1.2.5-0.fc29.x86_64
libsss_sudo-2.0.0-5.fc29.x86_64
libfontenc-1.1.3-9.fc29.x86_64
<snip>
```

Add the **sort** and **uniq** commands to sort the list and print the unique ones (since it's possible that some RPMs with identical names are installed):

```
[student@studentvm1 testdir]$ rpm -qa | sort | uniq
a2ps-4.14-39.fc29.x86_64
aajohan-comfortaa-fonts-3.001-3.fc29.noarch
abattis-cantarell-fonts-0.111-1.fc29.noarch
abiword-3.0.2-13.fc29.x86_64
abrt-2.11.0-1.fc29.x86_64
abrt-addon-ccpp-2.11.0-1.fc29.x86_64
abrt-addon-coredump-helper-2.11.0-1.fc29.x86_64
abrt-addon-kerneloops-2.11.0-1.fc29.x86_64
abrt-addon-pstoreoops-2.11.0-1.fc29.x86_64
abrt-addon-vmcore-2.11.0-1.fc29.x86_64
<snip>
```

Since this gives the correct list of RPMs you want to look at, you can use this as the input list to a loop that will print all the details of each RPM:

```
[student@studentvm1 testdir]$ for RPM in `rpm -qa | sort |
uniq` ; do rpm -qi $RPM ; done
```

This code produces way more data than you want. Note that the loop is complete. The next step is to extract only the information the PHBs requested. So, add an **egrep** command, which is used to select **^Name** or **^Summary**. The carat (**^**) specifies the beginning of the line; thus, any line with Name or Summary at the beginning of the line is displayed.

```
[student@studentvm1 testdir]$ for RPM in `rpm -qa | sort |
uniq` ; do rpm -qi $RPM ; done | egrep -i "^Name|^Summary"
Name        : a2ps
Summary     : Converts text and other types of files to
PostScript
Name        : aajohan-comfortaa-fonts
Summary     : Modern style true type font
Name        : abattis-cantarell-fonts
Summary     : Humanist sans serif font
Name        : abiword
Summary     : Word processing program
Name        : abrt
Summary     : Automatic bug detection and reporting tool
<snip>
```

You can try **grep** instead of **egrep** in the command above, but it will not work. You could also pipe the output of this

command through the **less** filter to explore the results. The final command sequence looks like this:

```
[student@studentvm1 testdir]$ for RPM in `rpm -qa | sort |
  uniq` ; do rpm -qi $RPM ; done | egrep -i "^Name|^Summary" >
  RPM-summary.txt
```

This command-line program uses pipelines, redirection, and a **for** loop—all on a single line. It redirects the output of your little CLI program to a file that can be used in an email or as input for other purposes.

This process of building up the program one step at a time allows you to see the results of each step and ensure that it is working as you expect and provides the desired results.

From this exercise, the PHBs received a list of over 1,900 separate RPM packages. I seriously doubt that anyone read that list. But I gave them exactly what they asked for, and I never heard another word from them about it.

Other loops

There are two more types of loop structures available in Bash: the **while** and **until** structures, which are very similar to each other in both syntax and function. The basic syntax of these loop structures is simple:

```
while [ expression ] ; do list ; done
```

and

```
until [ expression ] ; do list ; done
```

The logic of the first reads: "While the expression evaluates as true, execute the list of program statements. When the expression evaluates as false, exit from the loop." And the second: "Until the expression evaluates as true, execute the list of program statements. When the expression evaluates as true, exit from the loop."

While loop

The **while** loop is used to execute a series of program statements while (so long as) the logical expression evaluates as true. Your PWD should still be **~/testdir**.

The simplest form of the **while** loop is one that runs forever. The following form uses the true statement to always generate a "true" return code. You could also use a simple "1"—and that would work just the same—but this illustrates the use of the true statement:

```
[student@studentvm1 testdir]$ X=0 ; while [ true ] ; do echo
  $X ; X=$((X+1)) ; done | head
0
1
2
3
4
5
6
7
8
9
[student@studentvm1 testdir]$
```

This CLI program should make more sense now that you have studied its parts. First, it sets **$X** to zero in case it has a value left over from a previous program or CLI command. Then, since the logical expression **[true]** always evaluates to 1, which is true, the list of program instructions between **do** and **done** is executed forever—or until you press **Ctrl+C** or otherwise send a signal 2 to the program. Those instructions are an arithmetic expansion that prints the current value of **$X** and then increments it by one.

One of the tenets of *The Linux Philosophy for Sysadmins* [2] is to strive for elegance, and one way to achieve elegance is simplicity. You can simplify this program by using the variable increment operator, **++**. In the first instance, the current value of the variable is printed, and then the variable is incremented. This is indicated by placing the **++** operator after the variable:

```
[student@studentvm1 ~]$ X=0 ; while [ true ] ; do echo $((X++)) ;
  done | head
0
1
2
3
4
5
6
7
8
9
```

Now delete l **head** from the end of the program and run it again.

In this version, the variable is incremented before its value is printed. This is specified by placing the **++** operator before the variable. Can you see the difference?

```
[student@studentvm1 ~]$ X=0 ; while [ true ] ; do echo $((++X)) ;
  done | head
1
2
3
4
5
6
7
8
9
```

You have reduced two statements into a single one that prints the value of the variable and increments that value. There is also a decrement operator, **--**.

You need a method for stopping the loop at a specific number. To accomplish that, change the true expression to an actual numeric evaluation expression. Have the program loop to 5 and stop. In the example code below, you can see that **-le** is the logical numeric operator for "less than or equal to." This means: "So long as **$X** is less than or equal to 5, the loop will continue. When **$X** increments to 6, the loop terminates."

```
[student@studentvm1 ~]$ X=0 ; while [ $X -le 5 ] ; do echo
  $((X++)) ; done
0
1
2
3
4
5
[student@studentvm1 ~]$
```

Until loop

The **until** command is very much like the **while** command. The difference is that it will continue to loop until the logical expression evaluates to "true." Look at the simplest form of this construct:

```
[student@studentvm1 ~]$ X=0 ; until false  ; do echo $((X++)) ;
  done | head
0
1
2
3
4
5
6
7
8
9
[student@studentvm1 ~]$
```

It uses a logical comparison to count to a specific value:

```
[student@studentvm1 ~]$ X=0 ; until [ $X -eq 5 ]  ; do echo
  $((X++)) ; done
0
1
2
3
4
[student@studentvm1 ~]$ X=0 ; until [ $X -eq 5 ]  ; do echo
  $((++X)) ; done
1
2
3
4
5
[student@studentvm1 ~]$
```

Summary

This guide has explored many powerful tools for building Bash command-line programs and shell scripts. But it has barely scratched the surface on the many interesting things you can do with Bash; the rest is up to you.

I have discovered that the best way to learn Bash programming is to do it. Find a simple project that requires multiple Bash commands and make a CLI program out of them. Sysadmins do many tasks that lend themselves to CLI programming, so I am sure that you will easily find tasks to automate.

Many years ago, despite being familiar with other shell languages and Perl, I made the decision to use Bash for all of my sysadmin automation tasks. I have discovered that—sometimes with a bit of searching—I have been able to use Bash to accomplish everything I need.

Links

[1] http://www.both.org/?page_id=1183
[2] https://www.apress.com/us/book/9781484237298