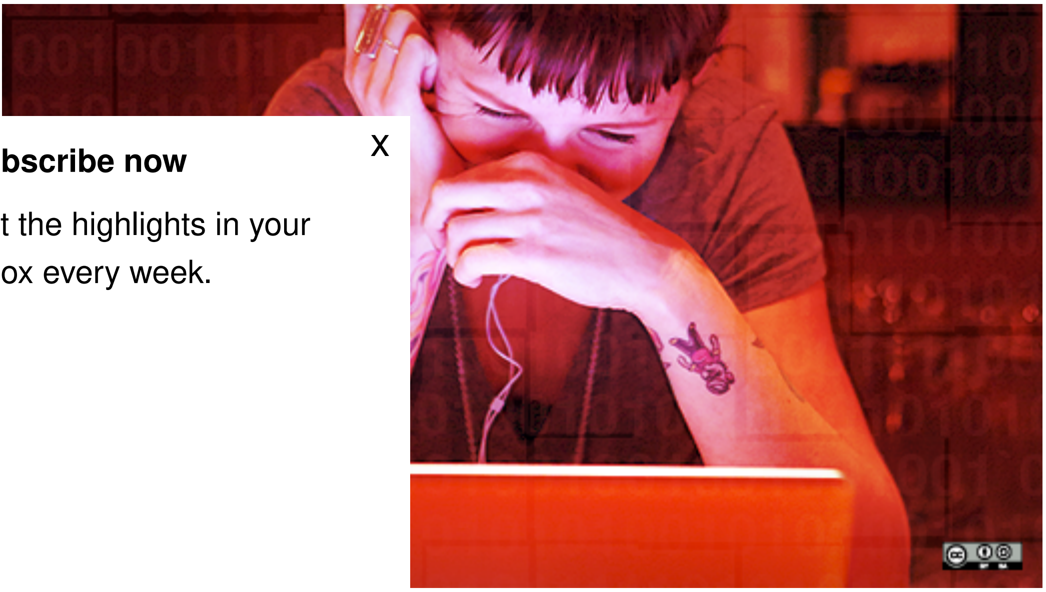
am with Bash: ators and shell expansions

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

22 Oct 2019 | David Both (Correspondent) (/users/dboth) | 122 | 3 comments



g language, one perfectly designed for use on the command line and in shell scripts. This three-part series (which is based on my three-volume Linux self-study course (http://www.both.org/?page_id=1183)) explores using Bash as a programming language on the command-line interface (CLI).

The first article (https://opensource.com/article/19/10/programming-bash-part-1) explored some simple command-line programming with Bash, including using variables and control operators. This second article looks into the types of file, string, numeric, and miscellaneous logical operators that provide execution-flow control logic and different types of shell expansions in Bash. The third and final article in the series will explore the **for**, **while**, and **until** loops that enable repetitive operations.

Logical operators are the basis for making decisions in a program and

Logical operators

Bash has a large set of logical operators that can be used in conditional
rm of the **if** control structure tests for a
ist of program statements if the condition is
perators: file, numeric, and non-numeric
ıs true (0) if the condition is met and false (1)

comparison operators is one or two
t are placed within square braces, followed
that are executed if the condition is true, and
ements if the condition is false:

```
                                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 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.

	Description
	le exists; it can be empty or have some so long as it exists, this will be true
	le exists and is a block special file such as a ke /**dev**/**sda** or /**dev**/**sda1**
	le exists and is a character special file such evice like /**dev**/**TTY1**
	le 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
	True if the file exists and has a size greater than zero; a

Operator	Description
	terminal
	le exists and its **set-user-id** bit is set
	le exists and is writable
	le exists and is executable
	le exists and is owned by the effective
	le exists and is a symbolic link
	le exists and has been modified since it was
-O filename	le 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

```
The file TestFile1 does not exist.
[student@studentvm1 testdir]$
```

med **TestFile1**. For now, it does not need to

```
$ touch TestFile1
```

f the **$File** variable rather than a text string
ations in this short CLI program:

```
$ File="TestFile1" ; if [ -e $File ] ; th

$
```

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 $File ] ; th
[student@studentvm1 testdir]$
```

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 $
TestFile1 exists and contains data.
```

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:

```
$ File="TestFile1" ; rm $File ; if [ -s $I
is empty.
```

t:

```
$ File="TestFile1" ; touch $File ; if [ -
is empty.
```

d test again:

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

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

```
File="TestFile1"
echo "This is $File" > $File
if [ -s $File ]

                                    ntains data."


                                     empty."

                                    t."
```

gram rewritten as it would appear in a

y for most CLI programs. Although any Linux / 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

Operator	Description
	conformance. When used with the **[[** command, rforms pattern matching as described above ound commands).
	the strings are not equal
	string1 sorts before string2 graphically (refers to locale-specific sorting nces for all alphanumeric and special ters)
	string1 sorts after string2 lexicographically

erators

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 "My
MyVar is zero length.
[student@studentvm1 testdir]$ MyVar="Random text" ; if [ -z "" ] ; th
MyVar is zero length.
```

You could also do it this way:

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

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

out what it can do. Note that quotes are

ariable you're testing.

```
$ MyVar="" ; expr length "$MyVar"

$ MyVar="How long is this?" ; expr length

$ expr length "We can also find the lengt
```

rs, I use a lot of testing in my scripts to

are equal (i.e., identical). I use the non-

on operator:

```
[student@studentvm1 testdir]$ Var1="Hello World" ; Var2="Hello World"
Var1 matches Var2
[student@studentvm1 testdir]$ Var1="Hello World" ; Var2="Hello world"
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

Operator	Description
arg1 -lt arg2	True if arg1 is less than arg2
	is less than or equal to arg2
	is greater than arg2
	is greater than or equal to arg2

ison logical operators

s. The first instance sets the variable **$X** to

al to 1. In the second instance, **X** is set to 0,

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

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

Operator	Description
	Bash set builtin in the Bash man page)
	hell variable varname is set (has been value)
	hell variable varname is set and is a name

gical operators

out these operators.

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 article 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.

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

it? But look what happens when you use it

```
$ echo "Hello "{David,Jen,Rikki,Jason}.
o Rikki. Hello Jason.
```

ul—it could save a good deal of typing. Now

```
$ echo b{ed,olt,ar}s
```

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 $Var
```

Pathname expansion

Pathname expansion is a fancy term expanding file-globbing patterns, using

full names of directories that match the

special pattern characters that enable

file names, directories, and other strings

ıs. These special pattern characters allow

ecific characters in a string.

y character in the specified location within

f any character in the specified location within

atching 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
chapter7  cpuHog.Linux  dmesg2.txt  Downloads  Pictures    Templates
testdir  cpuHog.mac    dmesg3.txt  file005    Public      testdir
cpuHog     Desktop       dmesg.txt   link3      random.txt  testdir1
[student@studentvm1 testdir]$
```

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

```
Documents:
```

```
file03       file11  file19       test06  test14  testfile09  testfi
file04       file12  file20       test07  test15  testfile10  testfi
file05       file13  Student1.txt test08  test16  testfile11  testfi
file06       file14  test01       test09  test18  testfile12  testfi
```

```
$
```

vanted. It listed the contents of the directories the directories and not their contents, use the

```
$ ls -d Do*

$
```

xpands 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;

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

sh takes on its meaning as a special
ırenthetical form uses only single
 the command statement.

n command-line programs and scripts where
n be used as an argument for another

le that uses both forms of this expansion
e PWD):

```
$ 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
```

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

it more useful, like creating a large number of

```
|$ for I in $(seq -w 5000) ; do touch file
```

q -w 5000 generates a list of numbers from
ıd substitution as part of the **for** statement,
ıe **for** statement to generate the numerical

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)) ;
Var 3 = 35
```

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

ften do in a script or CLI program that tells
ıory I have in a Linux host. The **free**
t data:

```
$ RAM=`free | grep ^Mem | awk '{print $2}
1452
8532
```

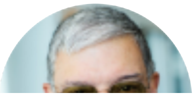
the sections of code used for command

n mostly for checking system resource
amounts in a script and then choose a program execution path based on the result.

Summary

This article, the second in this series 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 article in this series will explore the use of loops for performing various types of iterative operations.



About the author

David Both - David Both is an Open Source Software and GNU/Linux

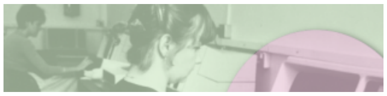
iter, and speaker who lives in Raleigh North Carolina.
ınent of and evangelist for the "Linux Philosophy." David
ıdustry for nearly 50 years. He has taught RHCE
and has worked at MCI Worldcom, Cisco, and the State
e has been working with Linux and Open Source
ı years. David prefers to purchase the components and

ısers/dboth)

Why I still love tcsh after all these years (/article /20/8/tcsh?utm_campaign=intrel)

Read and write data from anywhere with redirection in the Linux terminal (/article /20/6/redirection-bash?utm_campaign=intrel)

Using Bash traps in your scripts (/article /20/6/bash-trap?utm_campaign=int



Make Bash history more useful with these tips (/article

Take control of your data with associative arrays in Bash (/article
'0/6/associative-
arrays-

How to use Bash history commands (/article/20/6/bash-
history-
commands?utm_campa

Subscribe now

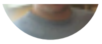
X

Get the highlights in your inbox every week.

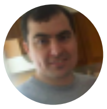
ɔbravo) on 22 Oct 2019 2

ınks for share!

lanfdoss) on 22 Oct 2019 2



Great article! This might become one of my goto (no pun intended) resources when I'm scripting.



Alan Formy-Duval (/users/alanfdoss) on 23 Oct 2019 2

One more thing, regarding command substitution, I've always used the older form of `command` but I've discovered that created problems when writing documents in Markdown, since the ticks are used to denote code, like:
`# ls -l`
So, switching to the newer form of $(command) resolves this formatting conflict.