

LINUX

Shell Scripting II

AUGUST 2025



String Manipulation

- Let's go deeper and find out how to work with strings in scripts.
- A string variable contains a sequence of text characters. It can include letters, numbers, symbols and punctuation marks. Some examples include: `abcde`, `123`, `abcde 123`, `abcde-123`, `&acbd e=%123`.
- String operators include those that do comparison, sorting, and finding the length. The following table demonstrates the use of some basic string operators:
- Remember, in most cases, we can use single square brackets (`[]`) instead of double (`[[]]`) in comparisons and logical tests, but the more modern doubled form helps avoid some errors, such as those that can arise when doing a comparison with empty strings and environment variables.

OPERATOR	MEANING
<code>[[string1 > string2]]</code>	Compares the sorting order of string1 and string2
<code>[[string1 == string2]]</code>	Compares the characters in string1 with the characters in string2
<code>myLen1=\${#string1}</code>	Saves the length of string1 in the variable myLen1

Example of String Manipulation

➤ In the first example, we compare two strings and display an appropriate message using the if statement. In the results shown, note the third test where there is an error if we use single brackets and do not put the variable name in quotes.

➤ Comparing strings and Using the if Statement

➤ In the second example, we pass in a file name and see if that file exists or not.

```
coop@r9:/tmp
r9:/tmp>cat file_test.sh
#!/bin/bash

echo "Give a file name to check to see if it exists"
read filename

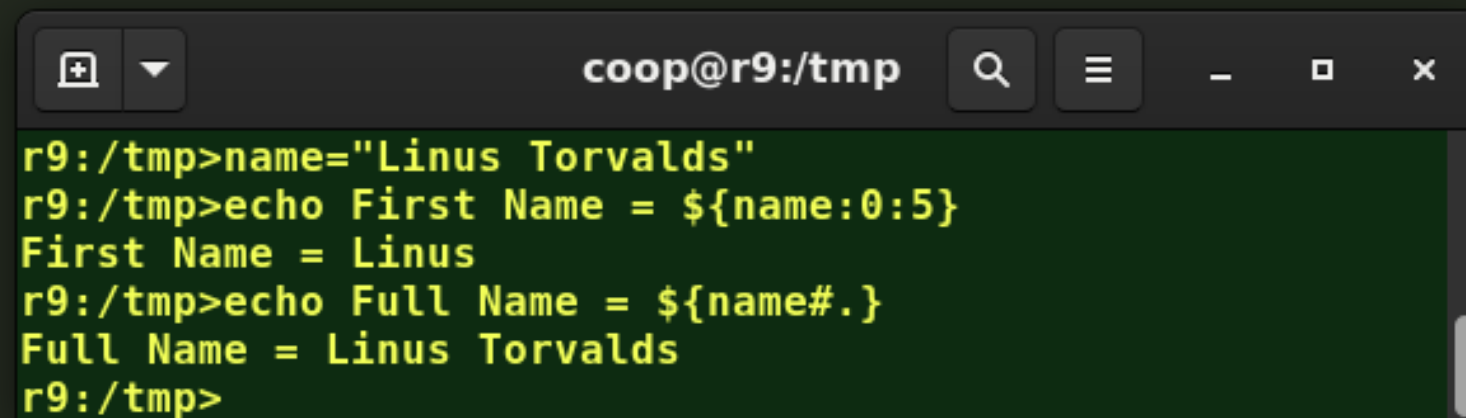
if [[ -f "$filename" ]] ; then
    echo "$filename exists"
else
    echo "$filename does not exist"
fi

r9:/tmp>./file_test.sh
Give a file name to check to see if it exists
/etc/passwd
/etc/passwd exists
r9:/tmp>./file_test.sh
Give a file name to check to see if it exists
/etc/malware
/etc/malware does not exist
r9:/tmp>
```

```
coop@r9:/tmp
r9:/tmp>./string_comp.sh
Give two strings to compare
hello goodbye
str1=hello, str2=goodbye
Testing with double brackets, no quotes
These strings are not the same
Testing with single brackets, with quotes
These strings are not the same
Testing with single brackets, no quotes
These strings are not the same
r9:/tmp>./string_comp.sh
Give two strings to compare
hello hello
str1=hello, str2=hello
Testing with double brackets, no quotes
These strings are identical
Testing with single brackets, with quotes
These strings are identical
Testing with single brackets, no quotes
These strings are identical
r9:/tmp>./string_comp.sh
Give two strings to compare
hello
str1=hello, str2=
Testing with double brackets, no quotes
These strings are not the same
Testing with single brackets, with quotes
These strings are not the same
Testing with single brackets, no quotes
./string_comp.sh: line 24: [: hello: unary operator expected
These strings are not the same
r9:/tmp>
```

Parts of a String

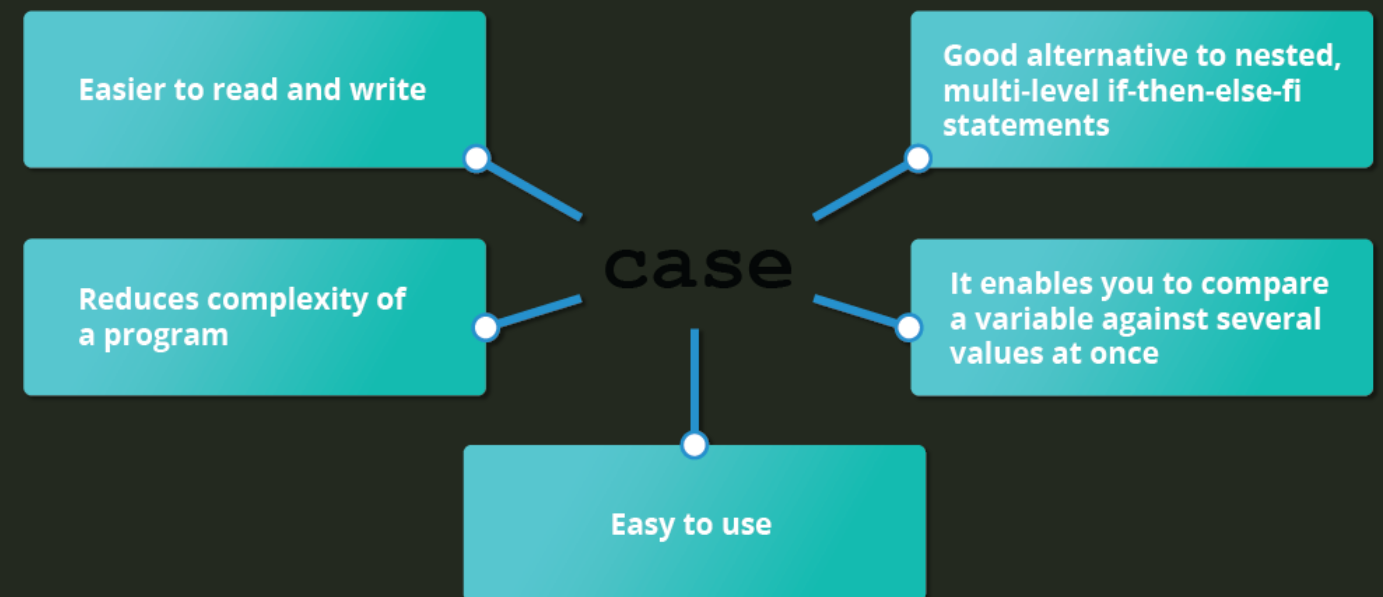
- ↘ At times, you may not need to compare or use an entire string. To extract the first *n* characters of a string, we can specify: `${string:0:n}`. Here, 0 is the offset in the string (i.e., which character to begin from) where the extraction needs to start, and *n* is the number of characters to be extracted.
- ↘ To extract all characters in a string after a dot (.), use the following expression: `${string#*.}`.



```
coop@r9:/tmp
r9:/tmp>name="Linus Torvalds"
r9:/tmp>echo First Name = ${name:0:5}
First Name = Linus
r9:/tmp>echo Full Name = ${name#*.}
Full Name = Linus Torvalds
r9:/tmp>
```

The case Statement

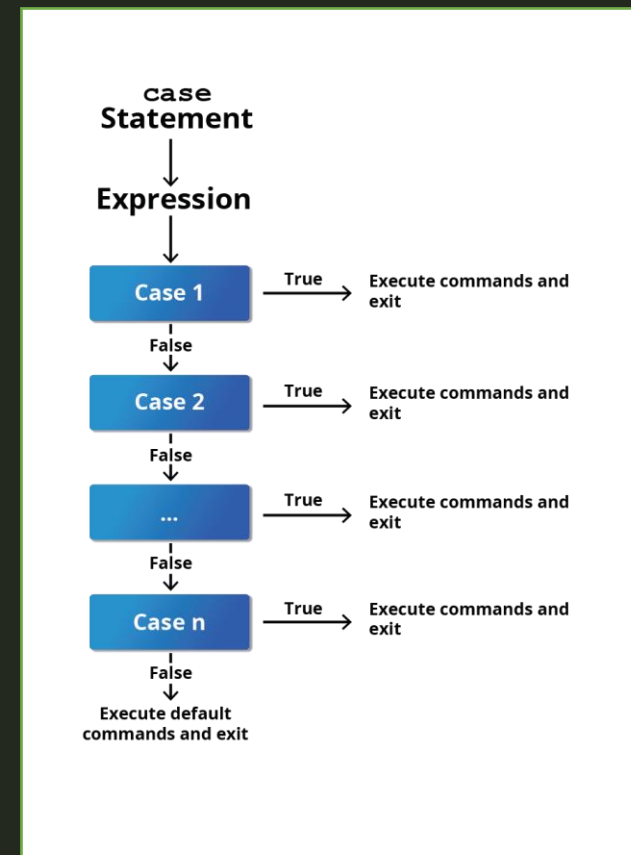
- ✚ The case statement is used in scenarios where the actual value of a variable can lead to different execution paths. case statements are often used to handle command-line options.
- ✚ Below are some of the advantages of using the case statement:
- ✚ It is easier to read and write and is a good alternative to nested, multi-level if-then-else-fi code blocks.
- ✚ It enables you to compare a variable against several values at once.
- ✚ It reduces the complexity of a program.



Structure of the case Statement

➤ Here is the basic structure of the case statement:

```
➤ case expression in  
  pattern1) execute commands;;  
  pattern2) execute commands;;  
  pattern3) execute commands;;  
  pattern4) execute commands;;  
  * )      execute some default commands or nothing ;;  
esac
```



➤ Note that as soon as the expression matches a pattern successfully, the execution path exits; i.e., the further tests are neither executed nor evaluated. If none of the tests return success, the final choice will execute, which can be to do nothing.

Example of Use of the case Construct

➤ Here is an example of the use of a case construct. Note you can have multiple possibilities for each case value that take the same action.

```
coop@r9:/tmp
r9:/tmp>cat testcase.sh
#!/bin/sh

echo "Do you want to destroy your entire file system?"
read response

case "$response" in
    "yes")          echo "I hope you know what you are doing!" ;;
    "no" )          echo "You have some comon sense!" ;;
    "y" | "Y" | "YES" ) echo "I hope you know what you are doing!" ;
                    echo 'I am supposed to type: " rm -rf /"' ;
                    echo "But I am not going to let you commit suicide";;
    "n" | "N" | "NO" ) echo "You have some common sense!" ;;
    * )             echo "You have to give an answer!" ;;
esac
exit 0
r9:/tmp>./testcase.sh
Do you want to destroy your entire file system?
N
You have some common sense!
r9:/tmp>./testcase.sh
Do you want to destroy your entire file system?

You have to give an answer!
r9:/tmp>./testcase.sh
Do you want to destroy your entire file system?
YES
I hope you know what you are doing!
I am supposed to type: " rm -rf /"
But I am not going to let you commit suicide
r9:/tmp>
```

Looping Constructs

- By using looping constructs, you can execute one or more lines of code repetitively, usually on a selection of values of data such as individual files. Usually, you do this until a conditional test returns either true or false, as is required.
- Three frequently used types of loops are often used in bash and in many programming languages:
 - for
 - while
 - until
- All these loops are easily used for repeatedly executing one or more statements until the exit condition is true. Each has an easily understood structural form.

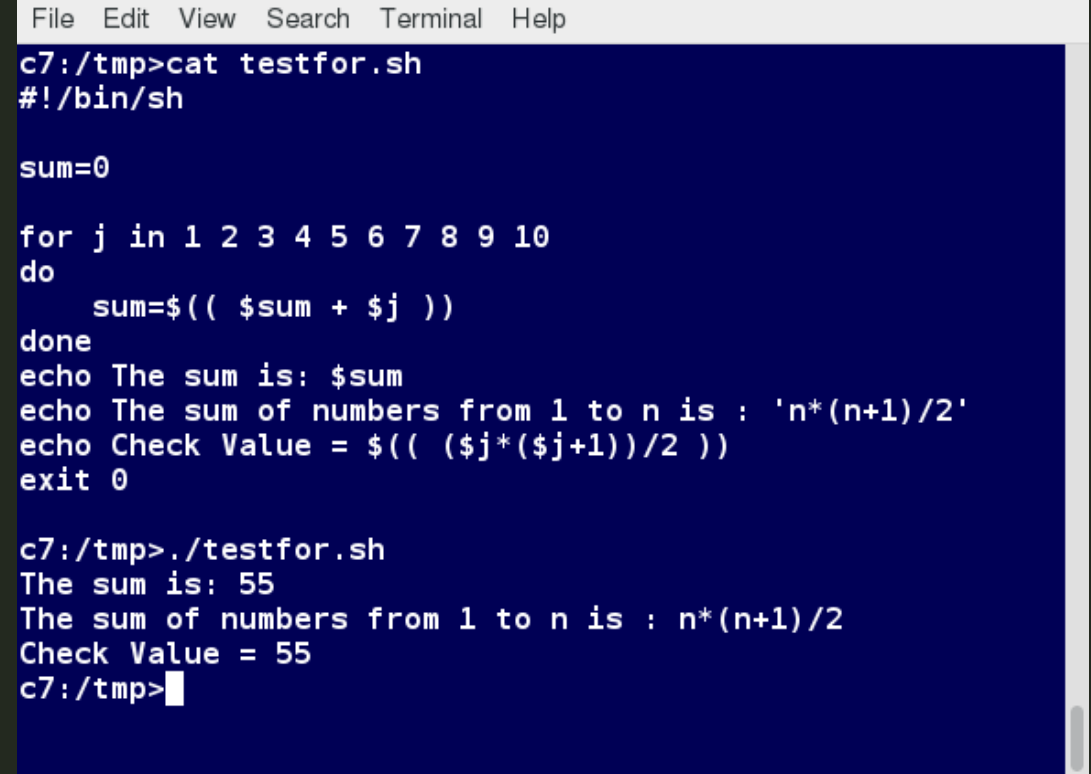
The for loop

➤ The for loop operates on each element of a list of items. The syntax for the for loop is:

```
➤ for variable-name in list
do
execute one iteration for each item in
the list until the list is finished
done
```

➤ In this case, variable-name and list are substituted by you as appropriate (see examples). As with other looping constructs, the statements that are repeated should be enclosed by do and done.

➤ The screenshot here shows an example of the for loop to print the sum of numbers 1 to 10.



```
File Edit View Search Terminal Help
c7:/tmp>cat testfor.sh
#!/bin/sh

sum=0

for j in 1 2 3 4 5 6 7 8 9 10
do
    sum=$(( $sum + $j ))
done
echo The sum is: $sum
echo The sum of numbers from 1 to n is : 'n*(n+1)/2'
echo Check Value = $(( ($j*($j+1))/2 ))
exit 0

c7:/tmp>./testfor.sh
The sum is: 55
The sum of numbers from 1 to n is : n*(n+1)/2
Check Value = 55
c7:/tmp>
```

The while Loop

↘ The while loop repeats a set of statements as long as the control command returns true. The syntax is:

↘ while condition is true

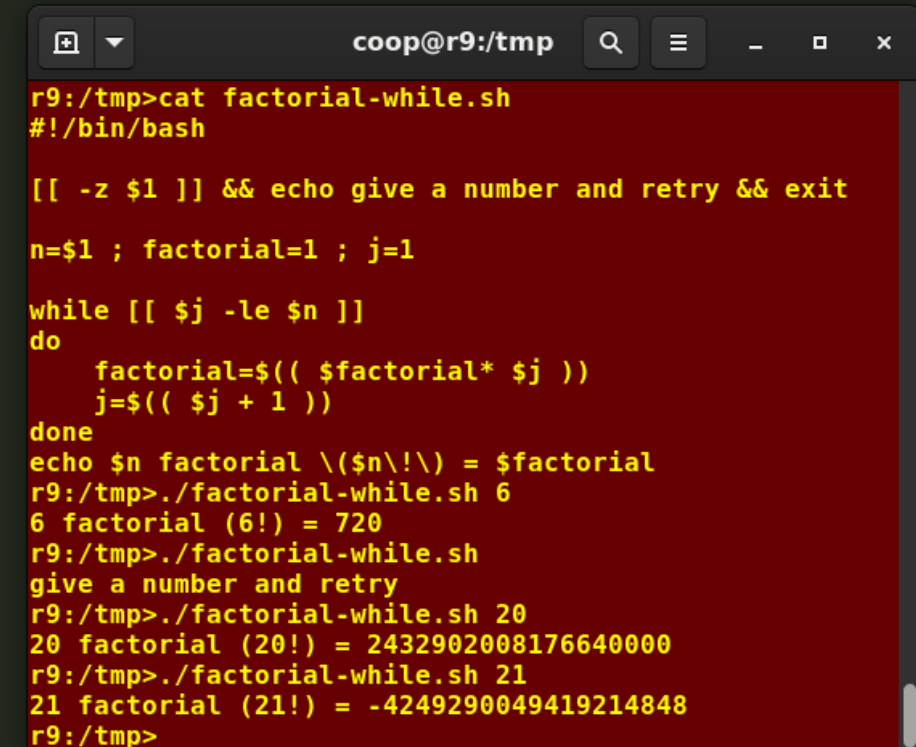
do

 Commands for execution

done

↘ The set of commands that need to be repeated should be enclosed between do and done. You can use any command or operator as the condition. Often, it is enclosed within square brackets ([]).

↘ The screenshot here shows an example of the while loop that calculates the factorial of a number. Do you know why the computation of 21! gives a bad result?



```
coop@r9:/tmp
r9:/tmp>cat factorial-while.sh
#!/bin/bash

[[ -z $1 ]] && echo give a number and retry && exit

n=$1 ; factorial=1 ; j=1

while [[ $j -le $n ]]
do
    factorial=$(( $factorial* $j ))
    j=$(( $j + 1 ))
done
echo $n factorial \($n!\) = $factorial
r9:/tmp>./factorial-while.sh 6
6 factorial (6!) = 720
r9:/tmp>./factorial-while.sh
give a number and retry
r9:/tmp>./factorial-while.sh 20
20 factorial (20!) = 2432902008176640000
r9:/tmp>./factorial-while.sh 21
21 factorial (21!) = -4249290049419214848
r9:/tmp>
```

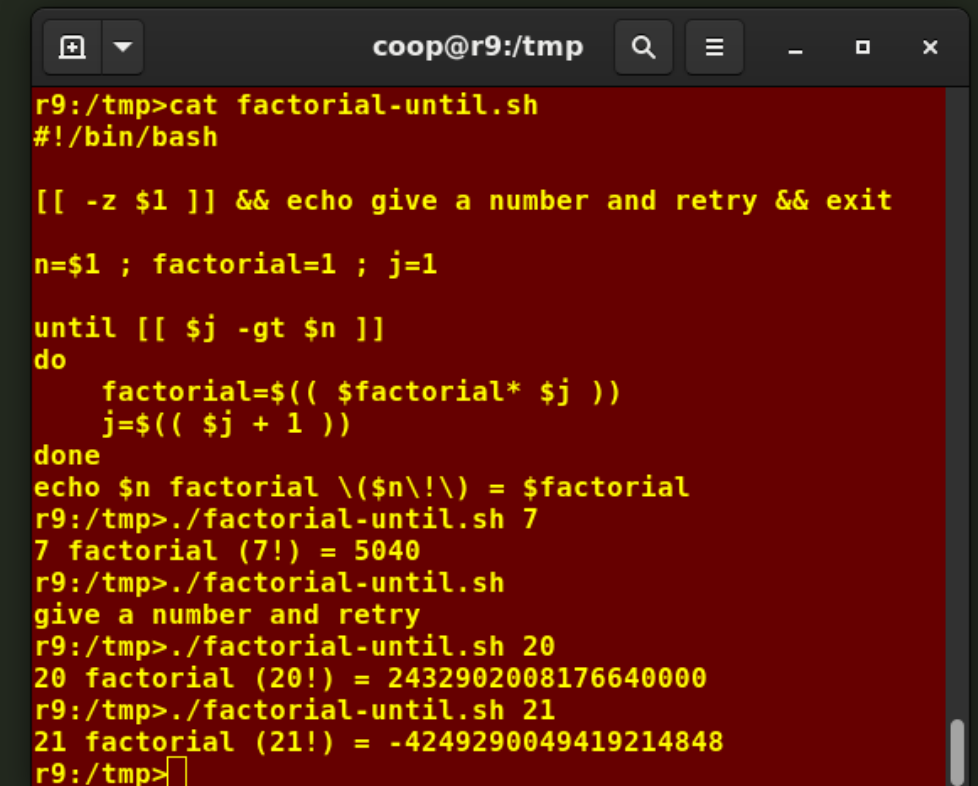
The until Loop

➤ The until loop repeats a set of statements as long as the control command is false. Thus, it is essentially the opposite of the while loop. The syntax is:

```
➤ until condition is false
do
  Commands for execution
----
done
```

➤ Similar to the while loop, the set of commands that need to be repeated should be enclosed between do and done. You can use any command or operator as the condition.

➤ The screenshot here shows an example of the until loop that once again computes factorials; it is only slightly different than the test case for the while loop.



```
coop@r9:/tmp
r9:/tmp>cat factorial-until.sh
#!/bin/bash

[[ -z $1 ]] && echo give a number and retry && exit

n=$1 ; factorial=1 ; j=1

until [[ $j -gt $n ]]
do
    factorial=$(( $factorial* $j ))
    j=$(( $j + 1 ))
done
echo $n factorial \($n\!\) = $factorial
r9:/tmp>./factorial-until.sh 7
7 factorial (7!) = 5040
r9:/tmp>./factorial-until.sh
give a number and retry
r9:/tmp>./factorial-until.sh 20
20 factorial (20!) = 2432902008176640000
r9:/tmp>./factorial-until.sh 21
21 factorial (21!) = -4249290049419214848
r9:/tmp>
```

Debugging bash Scripts

- While working with scripts and commands, you are likely to incur errors. These may be due to an error in the script, such as incorrect syntax, or other ingredients, such as a missing file or insufficient permission to do an operation. These errors may be reported with a specific error code but often yield incorrect or confusing output. So, how do you go about identifying and fixing an error?
- Debugging helps troubleshoot and resolve such errors and is one of the most important tasks a system administrator performs.

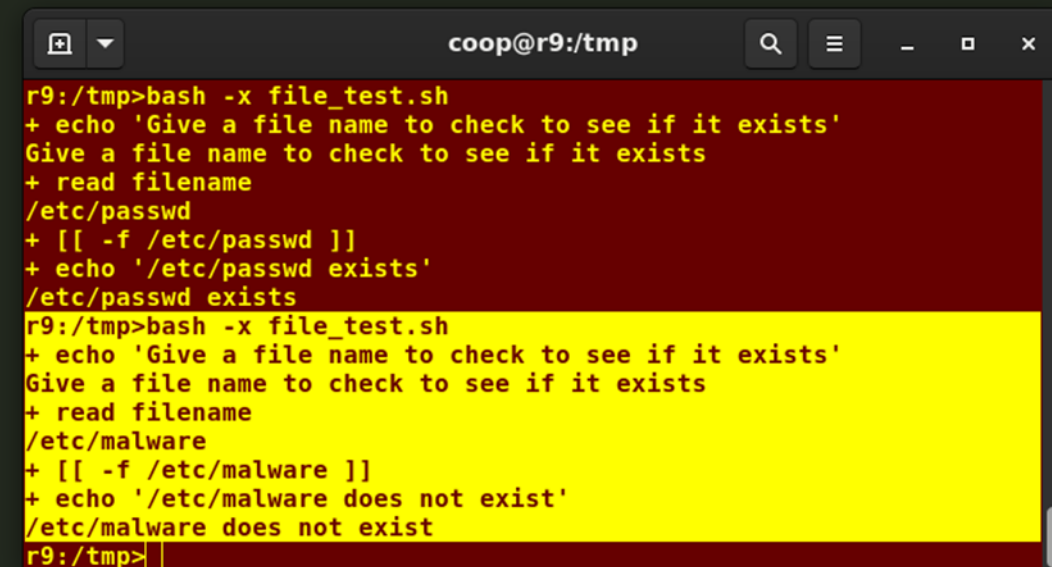
Script Debug Mode

- Before fixing an error (or bug), it is vital to locate the source.
- You can run a bash script in debug mode either by doing `bash -x ./script_file`, or bracketing parts of the script with `set -x` and `set +x`. The debug mode helps identify the error because:
 - It traces and prefixes each command with the `+` character.
 - It displays each command before executing it.
 - It can debug only selected parts of a script (if desired) with:

➤ `set -x` # turns on debugging

...

`set +x` # turns off debugging



```
r9:/tmp>bash -x file_test.sh
+ echo 'Give a file name to check to see if it exists'
Give a file name to check to see if it exists
+ read filename
/etc/passwd
+ [[ -f /etc/passwd ]]
+ echo '/etc/passwd exists'
/etc/passwd exists
r9:/tmp>bash -x file_test.sh
+ echo 'Give a file name to check to see if it exists'
Give a file name to check to see if it exists
+ read filename
/etc/malware
+ [[ -f /etc/malware ]]
+ echo '/etc/malware does not exist'
/etc/malware does not exist
r9:/tmp>
```

- The screenshot shown here demonstrates a previously demonstrated script that checks for a file's existence but now running in debug mode.

Redirecting Errors to File and Screen

- In UNIX/Linux, all programs that run are given three open file streams when they are started as listed in the table:
- By using redirection, we can save the standard output and error streams to one file or two separate files for later analysis after a program or command is executed.
- The screenshot shows a shell script with a simple bug, which is then run, and the error output is diverted to `error.log`. We then use `cat` to display the contents of the error log. Do you see how to fix the script?

FILE STREAM	DESCRIPTION	FILE DESCRIPTOR
stdin	Standard Input, by default the keyboard/terminal for programs run from the command line	0
stdout	Standard output, by default the screen for programs run from the command line	1
stderr	Standard error, where output error messages are shown or saved; by default, the same as stdout	2

```
student@openSUSE:/tmp
File Edit View Search Terminal Help
student@openSUSE:/tmp>
student@openSUSE:/tmp> cat testbasherror.sh
#!/bin/bash

sum=0
for i in 1 2 3 4
do
    sum=$((sum+$i))
done
echo "the sum of "$i" numbers is $sum"
ls error
student@openSUSE:/tmp> ./testbasherror.sh 2> error.log
student@openSUSE:/tmp> cat error.log
./testbasherror.sh: line 6: syntax error near unexpected token `('
./testbasherror.sh: line 6: `    sum=$((sum+$i))'
./testbasherror.sh: line 7: syntax error near unexpected token `done'
./testbasherror.sh: line 7: `done'
student@openSUSE:/tmp> 
```

Creating Temporary Files and Directories

- Consider a situation where you want to retrieve 100 lines from a file with 10,000 lines. You will need a place to store the extracted information, perhaps in a temporary file, while you do further processing on it.
- Temporary files (and directories) are meant to store data for a short time. Usually, one arranges it so that these files disappear when the program using them terminates. While you can also use touch to create a temporary file, in some circumstances, this may make it easy for hackers to gain access to your data. This is particularly true if the name and the file location of the temporary file are predictable.
- The best practice is to create random and unpredictable filenames for temporary storage. One way to do this is with the mktemp utility, as in the following examples.
- The XXXXXXXX is replaced by mktemp with random characters to ensure the name of the temporary file cannot be easily predicted and is only known within your program. You have to have at least 3 Xs in the supplied template, and the number of random characters will be equal to the number of Xs given.

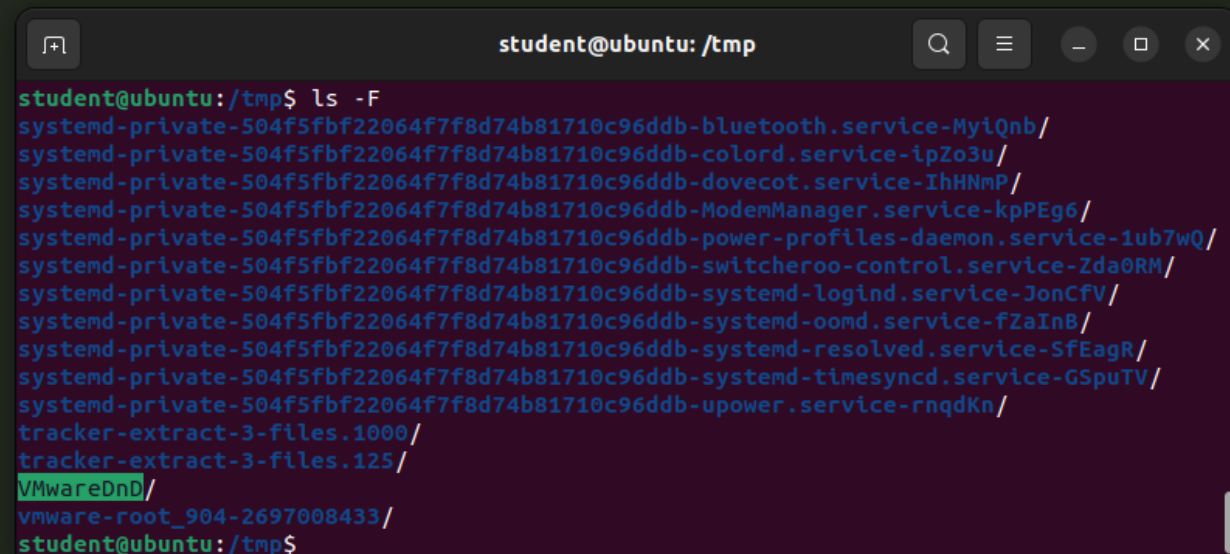
COMMAND	USAGE
<code>TEMP=\$(mktemp /tmp/tempfile.XXXXXXXX)</code>	To create a temporary file
<code>TEMPDIR=\$(mktemp -d /tmp/tempdir.XXXXXXXX)</code>	To create a temporary directory

Example of Creating a Temporary File and Directory

⚠ Sloppiness in creation of temporary files can lead to real damage, either by accident or if there is a malicious actor. For example, if someone were to create a symbolic link from a known temporary file used by root to the `/etc/passwd` file, like this:

⚠ `$ ln -s /etc/passwd /tmp/tempfile`

⚠ There could be a big problem if a script run by root has a line in it like this:

A terminal window titled 'student@ubuntu: /tmp' showing the output of the command 'ls -F'. The output lists various system service directories and files in /tmp, including systemd-private directories for services like bluetooth.service, colord.service, dovecot.service, ModemManager.service, power-profiles-daemon.service, switcheroo-control.service, systemd-logind.service, systemd-oond.service, systemd-resolved.service, systemd-timesyncd.service, and upower.service, as well as tracker-extract-3-files.1000/, tracker-extract-3-files.125/, VMwareDnD/, and vmware-root_904-2697008433/. The prompt is 'student@ubuntu: /tmp\$'.

⚠ `echo $VAR > /tmp/tempfile`

⚠ The password file will be overwritten by the temporary file contents.

⚠ To prevent such a situation, make sure you randomize your temporary file names by replacing the above line with the following lines:

⚠ `TEMP=$(mktemp /tmp/tempfile.XXXXXXXXXX)`

`echo $VAR > $TEMP`

⚠ The displayed screen capture shows similarly named temporary files on an Ubuntu system, which has randomly generated characters appended to their name.

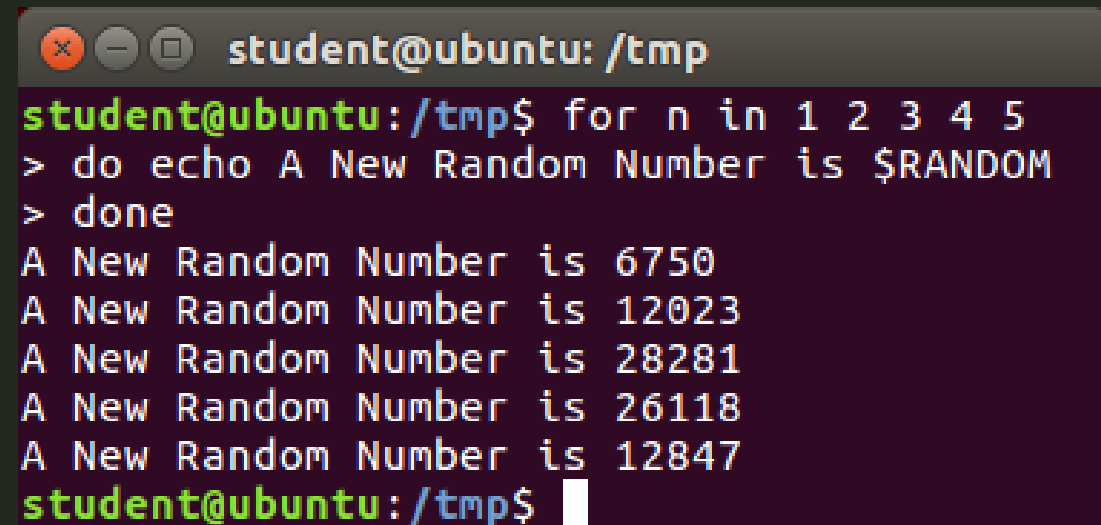
Discarding Output with /dev/null

- ↘ Certain commands (such as `find`) are quite capable of spewing voluminous and overwhelming amounts of output. To avoid this, we can redirect the large output to a special file (a device node) called `/dev/null`. This pseudofile is also called the bit bucket or black hole.
- ↘ All data written to `/dev/null` is discarded. Furthermore, write operations never return failure conditions. Using the proper redirection operators, it can make the uninteresting output disappear from commands that would normally generate output to `stdout` and/or `stderr`:
- ↘ `$ ls -lR /tmp > /dev/null`
- ↘ In the above command, the entire standard output stream is ignored, but any errors will still appear on the console. However, if one does:
- ↘ `$ ls -lR /tmp >& /dev/null`
- ↘ Both `stdout` and `stderr` will be dumped into `/dev/null`.

```
File Edit View Search Terminal Help
c7:/tmp>ls -lR > /dev/null
ls: cannot open directory ./pcp.iBMBphQI3: Permission denied
ls: cannot open directory ./systemd-private-dee8b3ebc6184b2aa5d894c4bd54
cfd0-colorD.service-nLrMij: Permission denied
ls: cannot open directory ./systemd-private-dee8b3ebc6184b2aa5d894c4bd54
cfd0-cups.service-Vm42q1: Permission denied
ls: cannot open directory ./systemd-private-dee8b3ebc6184b2aa5d894c4bd54
cfd0-httpd.service-9nSoUU: Permission denied
ls: cannot open directory ./vmware-root: Permission denied
c7:/tmp>
c7:/tmp>
c7:/tmp>ls -lR >& /dev/null
c7:/tmp>
```

Random Numbers and Data

- It is often useful to generate random numbers and other random data when performing tasks such as:
- Performing security-related tasks
- Reinitializing storage devices
- Erasing and/or obscuring existing data
- Generating meaningless data to be used for tests



```
student@ubuntu: /tmp
student@ubuntu:/tmp$ for n in 1 2 3 4 5
> do echo A New Random Number is $RANDOM
> done
A New Random Number is 6750
A New Random Number is 12023
A New Random Number is 28281
A New Random Number is 26118
A New Random Number is 12847
student@ubuntu:/tmp$
```

- Such random numbers can be generated by using the \$RANDOM environment variable, which is derived from the Linux kernel's built-in random number generator, or by the OpenSSL library function, which uses the FIPS140 (Federal Information Processing Standard) algorithm to generate random numbers for encryption
- To learn about FIPS140, read Wikipedia's "FIPS 140-2" article.
- The example shows you how to easily use the environmental variable method to generate random numbers

How the Kernel Generates Random Numbers

- Some servers have hardware random number generators that take as input different types of noise signals, such as thermal noise and photoelectric effect. A transducer converts this noise into an electric signal, which is again converted into a digital number by an A-D converter. This number is considered random. However, most common computers do not contain such specialized hardware and, instead, rely on events created during booting to create the raw data needed.
- Regardless of which of these two sources is used, the system maintains a so-called **entropy pool** of these digital numbers/random bits. Random numbers are created from this entropy pool.
- The Linux kernel offers the **/dev/random** and **/dev/urandom** device nodes, which draw on the entropy pool to provide random numbers which are drawn from the estimated number of bits of noise in the entropy pool.
- /dev/random** is used where very high-quality randomness is required, such as a one-time pad or key generation, but it is relatively slow to provide values. **/dev/urandom** is faster and suitable (good enough) for most cryptographic purposes.
- Furthermore, when the entropy pool is empty, **/dev/random** is blocked and does not generate any number until additional environmental noise (network traffic, mouse movement, etc.) is gathered, whereas **/dev/urandom** reuses the internal pool to produce more pseudo-random bits.

```
File Edit View Search Terminal Help
c7:/tmp>ls -l /dev/*random
crw-rw-rw- 1 root root 1, 8 Dec 23 07:06 /dev/random
crw-rw-rw- 1 root root 1, 9 Dec 23 07:06 /dev/urandom
c7:/tmp>
```

Summary

- You can manipulate strings to perform actions such as comparison, sorting, and finding length.
- You can use Boolean expressions when working with multiple data types, including strings or numbers, as well as files.
- The output of a Boolean expression is either true or false.
- Operators used in Boolean expressions include the && (AND), || (OR), and ! (NOT) operators.
- We looked at the advantages of using the case statement in scenarios where the value of a variable can lead to different execution paths.
- Script debugging methods help troubleshoot and resolve errors.
- The standard and error outputs from a script or shell commands can easily be redirected into the same file or separate files to aid in debugging and saving results.
- Linux allows you to create temporary files and directories, which store data for a short duration, both saving space and increasing security.
- Linux provides several different ways of generating random numbers, which are widely used.

info@[aistudio.com.tr](mailto:info@aistudio.com.tr)

aistudio.com.tr

Thank you!

