Chapter 16

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**, **&acbde=%123**.

String operators include those that do comparison, sorting, and finding the length. The following table demonstrates the use of some basic string operators:

Operator	Meaning
`[[ string1 > string2 ]]`	Compares the sorting order of `string1` and `string2`.
`[[ string1 == string2 ]]`	Compares the characters in `string1` with the characters in `string2`.
`myLen1=${#string1}`	Saves the length of `string1` in the variable `myLen1`.

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.

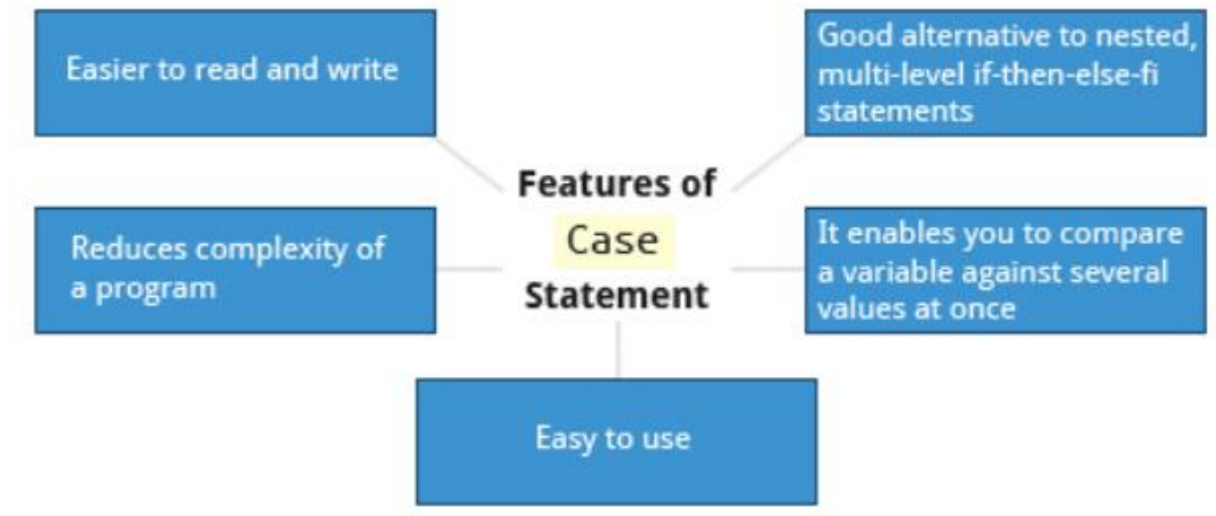
To extract all characters in a string after a dot (**.**), use the following expression: **${string#*.}**.

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.
- It 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.

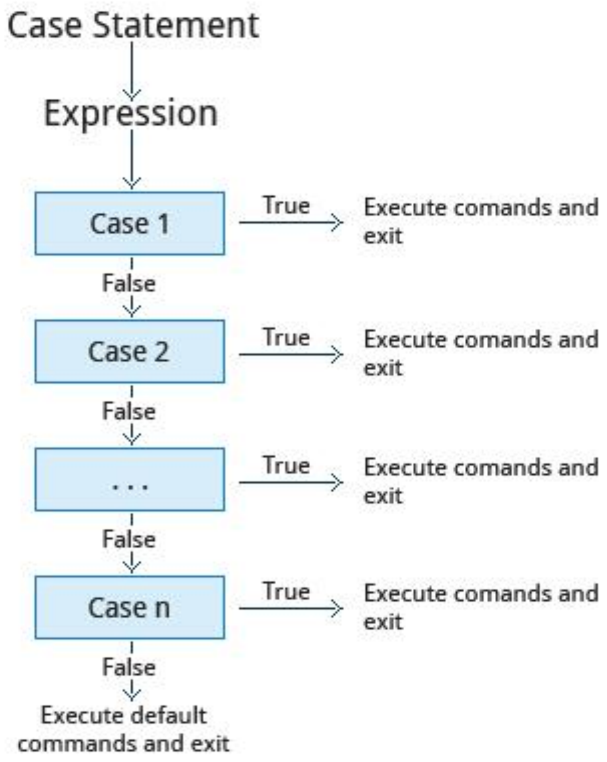


Easier to read and write
Good alternative to nested, multi-level if-then-else-fi statements
Features of Case Statement
Reduces complexity of a program
It enables you to compare a variable against several values at once
Easy to use

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
```



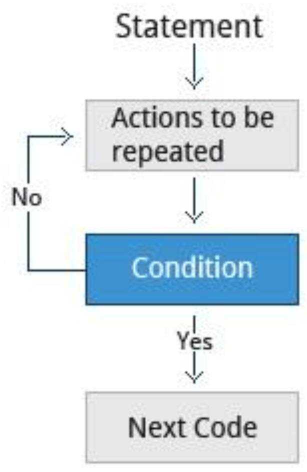
Looping constructs

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 type of loops are often used in most programming languages:

- **for**
- **while**
- **until**.

All these loops are easily used for repeating a set of statements until the exit condition is true.



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.

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?

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 example of the **until** loop that once again computes factorials; it is only slightly different than the test case for the **while** loop.

Debugging bash scripts

While working with scripts and commands, you may run into errors. These may be due to an error in the script, such as an 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 just yield incorrect or confusing output. So, how do you go about identifying and fixing an error?

Debugging helps you 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 know its source.

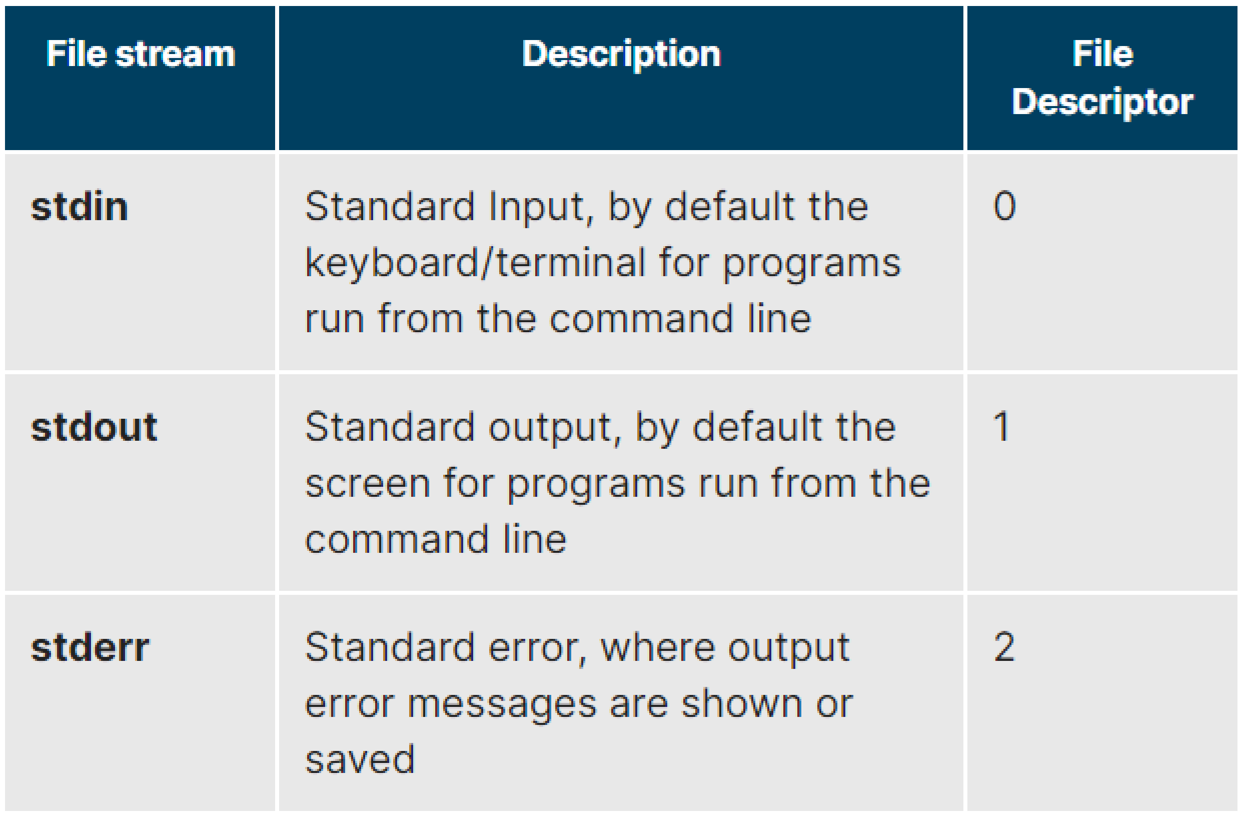
You can run a bash script in debug mode either by doing **bash –x ./script_file**, or bracketing parts of the script with **set -x** and **set +x**. The debug mode helps identify the error because:

- 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
```

The screenshot shown here demonstrates a script which runs in debug mode if run with any argument on the command line.

Redirecting errors to file and screen



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	2

Using redirection, we can save the stdout and stderr output 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**. Using **cat** to display the contents of the error log adds in debugging. Do you see how to fix the script?

Creating temporary files and directories

Consider a situation where you want to retrieve 100 records from a file with 10,000 records. 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.

Command	Usage
`TEMP=$(mktemp /tmp/tempfile.XXXXXXXX)`	To create a temporary file
`TEMPDIR=$(mktemp -d /tmp/tempdir.XXXXXXXX)`	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 like this:

```
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.XXXXXXXX)
echo $VAR > $TEMP
```

Note the screen capture shows similarly named temporary files from different days, but with randomly generated characters in them.

Discarding output with /dev/null

Certain commands (like **find**) will produce voluminous amounts of output, which can overwhelm the console. 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 /dev.

All data written to it is discarded and write operations never return a failure condition. Using the proper redirection operators, it can make the 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**.

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.

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 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.

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.