The if Statement

Conditional decision making, using an **if** statement, is a basic construct that any useful programming or scripting language must have.

When an **if** statement is used, the ensuing actions depend on the evaluation of specified conditions, such as:

* + - Numerical or string comparisons
    - Return value of a command (0 for success)
    - File existence or permissions.

In compact form, the syntax of an **if** statement is:

**if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi**

A more general definition is:

**if condition**  
**then**  
**statements**  
**else**  
**statements**  
**fi**

## Using the if Statement

n the following example, an **if** statement checks to see if a certain file exists, and if the file is found, it displays a message indicating success or failure:

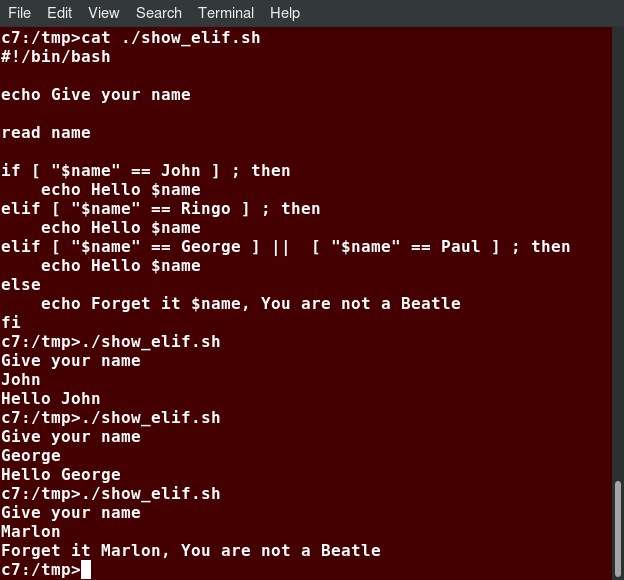
**if [ -f "$1" ]**  
**then**  
**echo file "$1 exists"**  
**else**  
**echo file "$1" does not exist**  
**fi**

We really should also check first that there is an argument passed to the script (**$1**) and abort if not.

Notice the use of the square brackets (**[]**) to delineate the test condition. There are many other kinds of tests you can perform, such as checking whether two numbers are equal to, greater than, or less than each other and make a decision accordingly; we will discuss these other tests.

In modern scripts, you may see doubled brackets as in **[[ -f /etc/passwd ]]**. This is not an error. It is never wrong to do so and it avoids some subtle problems, such as referring to an empty environment variable without surrounding it in double quotes; we will not talk about this here.

**if [ sometest ] ; then**  
**echo Passed test1**  
**elif [ somothertest ] ; then**  
**echo Passed test2**  
**fi**



# Testing for Files

Bash provides a set of file conditionals, that can be used with the **if** statement, including those in the table.

You can use the **if**statement to test for file attributes, such as:

* + - File or directory existence
    - Read or write permission
    - Executable permission.

For example, in the following example:  
  
**if [ -x /etc/passwd ] ; then**  
**ACTION**  
**fi**

the **if** statement checks if the file **/etc/passwd** is executable, which it is not. Note the very common practice of putting:

**; then**

on the same line as the **if** statement.

You can view the full list of file conditions typing:

**man 1 test**.

|  |  |
| --- | --- |
| **Condition** | **Meaning** |
| **-e file** | Checks if the file exists. |
| **-d file** | Checks if the file is a directory. |
| **-f file** | Checks if the file is a regular file (i.e. not a symbolic link, device node, directory, etc.) |
| **-s file** | Checks if the file is of non-zero size. |
| **-g file** | Checks if the file has **sgid** set. |
| **-u file** | Checks if the file has **suid** set. |
| **-r file** | Checks if the file is readable. |
| **-w file** | Checks if the file is writable. |
| **-x file** | Checks if the file is executable. |

Boolean expressions evaluate to either TRUE or FALSE, and results are obtained using the various Boolean operators listed in the table.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operation** | **Meaning** |
| **&&** | **AND** | The action will be performed only if both the conditions evaluate to true. |
| **||** | **OR** | The action will be performed if any one of the conditions evaluate to true. |
| **!** | **NOT** | The action will be performed only if the condition evaluates to false. |

Note that if you have multiple conditions strung together with the **&&** operator, processing stops as soon as a condition evaluates to false. For example, if you have **A && B && C** and A is true but B is false, C will never be executed.

Likewise, if you are using the **||** operator, processing stops as soon as anything is true. For example, if you have **A || B || C** and A is false and B is true, you will also never execute C.

Boolean expressions return either TRUE or FALSE. We can use such expressions when working with multiple data types, including strings or numbers, as well as with files. For example, to check if a file exists, use the following conditional test:

**[ -e <filename> ]**  
  
Similarly, to check if the value of **number1** is greater than the value of **number2**, use the following conditional test:

**[ $number1 -gt $number2 ]**  
  
The operator **-gt** returns TRUE if **number1**is greater than **number2**.

You can use the **if** statement to compare strings using the operator **==** (two equal signs). The syntax is as follows:

**if [ string1 == string2 ] ; then  
   ACTION  
fi**

Note that using one **=** sign will also work, but some consider it deprecated usage. Let’s now consider an example of testing strings.

In the example illustrated here, the **if** statement is used to compare the input provided by the user and accordingly display the result.

## Numerical Tests

You can use specially defined operators with the **if** statement to compare numbers. The various operators that are available are listed in the table:

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| **-eq** | Equal to |
| **-ne** | Not equal to |
| **-gt** | Greater than |
| **-lt** | Less than |
| **-ge** | Greater than or equal to |
| **-le** | Less than or equal to |

The syntax for comparing numbers is as follows:  
  
**exp1 -op exp2**

## Arithmetic Expressions

Arithmetic expressions can be evaluated in the following three ways (spaces are important!):

* Using the **expr** utility  
  **expr**is a standard but somewhat deprecated program. The syntax is as follows:  
    
  **expr 8 + 8**  
  **echo $(expr 8 + 8)**
* Using the **$((...))**syntax   
  This is the built-in shell format. The syntax is as follows:  
    
  **echo $((x+1))**
* Using the built-in shell command **let**. The syntax is as follows:  
    
  **let x=(1 + 2); echo $x**

In modern shell scripts, the use of **expr** is better replaced with **var=$((...))**.

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.

## Parts of a String

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

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**

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 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 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 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.

## 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

In UNIX/Linux, all programs that run are given three open file streams when they are started as listed in the table:

|  |  |  |
| --- | --- | --- |
| **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 touchto 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 |

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 black hole.

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

t 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*](https://en.wikipedia.org/wiki/FIPS_140-2)*"* article.

The example shows you how to easily use the environmental variable method to generate random numbers.

for i in 1 2 3 4   
do echo A new random number is $RANDOM  
done

## 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.

## 