**SHELL scripting content**

**Chapter-1**

1. What is Shell Scripting
2. What is Kernal
3. What is Shell
4. Types of Shell
5. Features of Shell Scripts
6. **What is Shell Scripting**

**Shell Scripting** is an open-source computer program designed to be run by the Unix/Linux shell. Shell Scripting is a program to write a series of commands for the shell to execute. It can combine lengthy and repetitive sequences of commands into a single and simple script that can be stored and executed anytime which, reduces programming efforts.

This Shell Scripting tutorial helps to learn a basic understanding of the Linux/Unix shell scripting program to advanced concepts of Shell Scripting. This Shell Script tutorial designed for beginners and professionals who want to learn What is Shell Scripting? How shell scripting works, types of shell, and more.

1. **What is Kernal**
2. **What is Shell**

**Shell** is a UNIX term for an interface between a user and an operating system service. Shell provides users with an interface and accepts human-readable commands into the system and executes those commands which can run automatically and give the program’s output in a shell script.

An Operating is made of many components, but its two prime components are –

* Kernel
* Shell

A Kernel is at the nucleus of a computer. It makes the communication between the hardware and software possible. While the Kernel is the innermost part of an operating system, a shell is the outermost one.

A shell in a Linux operating system takes input from you in the form of commands, processes it, and then gives an output. It is the interface through which a user works on the programs, commands, and scripts. A shell is accessed by a terminal which runs it.

When you run the terminal, the Shell issues a command prompt (usually $), where you can type your input, which is then executed when you hit the Enter key. The output or the result is thereafter displayed on the terminal.

The Shell wraps around the delicate interior of an Operating system protecting it from accidental damage. Hence the name Shell.

This Unix/Linux Shell Script tutorial helps understand shell scripting basics to advanced levels.

1. **Types of Shell**

Linux provides a wide choice of shells; exactly what is available on the system is listed in /etc/shells. Typical choices are:

/bin/sh  
/bin/bash  
/bin/tcsh  
/bin/csh  
/bin/ksh  
/bin/zsh

Most Linux users use the default bash shell, but those with long UNIX backgrounds with other shells may want to override the default.

History of Command Shells sh was written by Steve Bourne at AT&T in 1977, and is often known as the Bourne Shell. All other shells are descended from it in some fashion and it is available on all systems that have a UNIX bloodline.

csh was written by Bill Joy at UC Berkeley and released in 1978.

The internal syntax is quite different than sh and is designed to resemble the C programming language, and hence the name.

tcsh was originally developed by Ken Greer at Carnegie Mellon University in the late 1970's; the t in tcsh stands for TENEX, an operating system that was used on some DEC PDP-10's. It has many additional features as compared with csh and on virtually all modern systems csh is just a link to tcsh.

ksh was written by David Korn at AT&T and appeared in 1982, and is often known as the Korn shell. It was designed to be a major upgrade to sh and is backward compatible with it, and brings in some of the features of tcsh, such as command line history recall. This shell has long been a favorite of many system administrators.

bash is a product of the GNU project and was created in 1987. It was designed as a major upgrade of sh; the name stands for Bourne Again Shell. It has full backward compatibility with sh and partial compatibility with ksh. On all Linux systems sh is just a link to bash, but scripts which are invoked as sh will only work without the bash extensions.

A similar relationship exists between csh and tcsh.

**5. Features of Shell Scripts**

1. Automate tasks and reduce errors

2. Combine long and repetitive sequence of commands into a single command

3. Share procedures among several users.

4. Provide a controlled interface to users

5. Create new commands using combination of utilities

6. Quick prototyping no need to compile

**Uses of Shell Script**

Typing a long sequence of commands at a terminal window can be complicated, time consuming, and error prone. By deploying shell scripts, using the command line becomes an efficient and quick way to launch complex sequences of steps. The fact that shell scripts are saved in a file also makes it easy to use them to create new script variations and share standard procedures with several users.

**Summary:**

1. Kernel is the nucleus of the operating systems, and it communicates between hardware and software
2. Shell is a program which interprets user commands through CLI like Terminal
3. The Bourne shell and the C shell are the most used shells in Linux
4. Linux Shell scripting is writing a series of command for the shell to execute
5. Shell variables store the value of a string or a number for the shell to read
6. Shell scripting in Linux can help you create complex programs containing conditional statements, loops, and functions
7. Basic Shell Scripting Commands in Linux: cat, more, less, head, tail, mkdir, cp, mv, rm, touch, grep, sort, wc, cut and, more.
8. Shell basic scripts
9. Shell Variables
10. Interactive Example Using Bash Scripts
11. Basic Syntax and Special Characters
12. Return Values
13. Script Parameters
14. Environment Variables
15. Command Substitution
16. Splitting Long Commands Over Multiple Lines
17. Putting Multiple Commands on a Single Line
18. Output Redirection
19. Input Redirection
20. **How to Write Shell Script in Linux/Unix**

**Shell Scripts** are written using text editors. On your Linux system, open a text editor program, open a new file to begin typing a shell script or shell programming, then give the shell permission to execute your shell script and put your script at the location from where the shell can find it.

Let us understand the steps in creating a Shell Script:

1. **Create a file** **using** a **vi** editor (or any other editor). Name script file with **extension .sh**
2. **Start** the script with **#! /bin/sh**
3. Write some code.
4. Save the script file as filename.sh
5. For **executing** the script type **bash filename.sh**

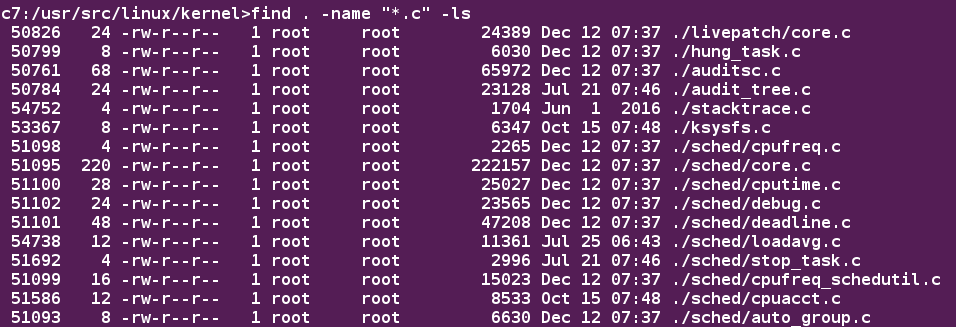
“#!” is an operator called **shebang** which directs the script to the interpreter location. So, if we use” #! /bin/sh” the script gets directed to the **bourne-shell.**

**#!**, contains the full path of the command interpreter (in this case **/bin/bash**) that is to be used on the file.

As we have noted, you have quite a few choices for the scripting language you can use, such as **/usr/bin/perl**, **/bi n/csh**, **/usr/bin/python**, etc.

For example, typing find . -name "\*.c" -ls at the command line accomplishes the same thing as executing a script file containing the lines:

**#! /bin/bash  
find . -name "\*.c" -ls**



Let's write a simple bash script that displays a one-line message on the screen. Either type:

**$ cat > hello.sh**  
**#!/bin/bash**  
**echo "Hello Linux Foundation Student"**

and press **ENTER** and **CTRL-D** to save the file, or just create **hello.sh** in your favourite text editor. Then, type **chmod +x hello.sh** to make the file executable by all users.

You can then run the script by typing **./hello.sh** or by doing:

**$ bash hello.sh  
Hello Linux Foundation Student**

$ cat hello.sh

$ bash hello.sh

$ chmod +x hello.sh

**script-1 => Shell1.sh**

mkdir dir1

cd dir1

touch file1

cd ..

touch file2

cd ..

sudo yum install vim -y

sudo yum install unzip -y

sudo yum install curl wget nano -y

sudo useradd vinodh

echo admin | passwd vinodh --stdin

// copy above example to root user home

// admin cmds like yum install \*\*\* will execute with root user only

// the meanig to this line is echo admin | passwd vinodh --stdin

# echo admin

admin

# passwd vinodh

changing password for user vinodh.

New password: XXXXX

**# chmod +x first.sh**

**script-2**

sudo useradd vinodh

echo admin | passwd vinodh --stdin

// the above example always creates vinodh user

// to this we use variables concept.

1. **What are Shell Variables?**

As discussed earlier, Variables store data in the form of characters and numbers. Similarly, Shell variables are used to store information and they can by the shell only.

For example, the following creates a shell variable and then prints it:

variable ="Hello"

echo $variable

Below is a small script which will use a variable.

#! /bin/sh

echo "what is your name?"

read name

echo "How do you do, $name?"

read remark

echo "I am $remark too!"

**script-3**

user="raju"

useradd $user

echo admin | passwd $user --stdin

chage -d 0 $user // immediately u hv to change the password upon user login.

//now you have to change at one place.

1. **Interactive Example Using bash Scripts**

Now, let's see how to create a more interactive example using a bash script. The user will be prompted to enter a value, which is then displayed on the screen.

The value is stored in a temporary variable, **name**.

We can reference the value of a shell variable by using a **$** in front of the variable name, such as **$name**.

To create this script, you need to create a file named **getname.sh** in your favourite editor with the following content:

**#!/bin/bash**  
**# Interactive reading of a variable**  
**echo "ENTER YOUR NAME"**  
**read name**  
**# Display variable input**  
**echo The name given was: $name**

Once again, make it executable by doing **chmod +x getname.sh**.

In the above example, when the user types **./getname.sh** and the script is executed, the user is prompted with the string **ENTER YOUR NAME**. The user then needs to enter a value and press the **Enter** key. The value will then be printed out.

***NOTE****: The hash-tag/pound-sign/number-sign (****#****) is used to start comments in the script and can be placed anywhere in the line (the rest of the line is considered a comment). However, note the special magic combination of****#!****, used on the first line, is a unique exception to this rule*

**Dynamically take values from keyboard**

- How to take values through bash cmd -e (extra)

# read -p "Enter the name:"

Enter the name: vinodh

# read -p "Enter the name " g

Enter the name: vinodh

# echo $g

vinodh

**$ cat getName.sh**

**$./getName.sh**

**script-4**

#!/bin/bash

read -p "Enter the username:" user

read -s -p "Enter the password" pass // -s -> silent

echo $pass | passwd $user --stdin

chage -d 0 $user

1. Special variables

// how to pass value while executing script from cmd

// # ./one.sh a b

$0 => Name of the script file itself

$1 => first arg

$2 => second arg

$3 => third arg

**$$** => 13402 => script id , on which id this script is executed.

**$#** => number of arguments

**$@** => prints the arguments

**$?** => 0 or 1 => repressed previously typed command (0 success 1 fail )

**script-5**

#!/bin/bash

# this is to explain special variables

#!/bin/bash

yum install $1 =y

systemctl start $1

systemctl enable $1

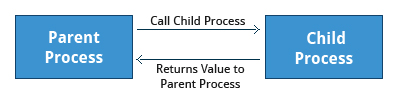
systemctl status $1

**# chmod +x shell5.sh httpd**

**# systemctl status httpd**

1. Return Values

All shell scripts generate a return value upon finishing execution, which can be explicitly set with the **exit** statement. Return values permit a process to monitor the exit state of another process, often in a parent-child relationship. Knowing how the process terminates enables taking any appropriate steps which are necessary or contingent on success or failure.



**Viewing Return Values**

As a script executes, one can check for a specific value or condition and return success or failure as the result. By convention, success is returned as **0**, and failure is returned as a non-zero value. An easy way to demonstrate success and failure completion is to execute ls on a file that exists as well as one that does not, the return value is stored in the environment variable represented by **$?**:

$ ls /etc/logrotate.conf  
/etc/logrotate.conf

$ echo $?  
0

In this example, the system is able to locate the file **/etc/logrotate.conf** and **ls** returns a value of **0** to indicate success. When run on a non-existing file, it returns **2**. Applications often translate these return values into meaningful messages easily understood by the user.

**$ clear**

**$ ls /etc/passwd**

**$ echo $?**

**$ ls /etc/passwdnot**

**$ echo $?**

Create a file named **testls.sh**, with the content below.

**#!/bin/bash**

**#**

**# check for non-existent file, exit status will be 2**

**#**

**ls SoMeFiLe.ext**

**echo "status: $?"**

**# create file, and do again, exit status will be 0**

**touch SoMeFiLe.ext**

**ls SoMeFiLe.ext**

**echo "status: $?"**

**# remove the file to clean up**

**rm SoMeFiLe.ext**

Make it executable and run it:

**student:/tmp> chmod +x testls.sh**

**student:/tmp> ./testls.sh**

**ls: cannot access SoMeFiLe.ext: No such file or directory**

**status: 2**

**SoMeFiLe.ext**

**status: 0**

**If echo $? Is return 0 then success, if it is 1 then it is failure.**

**#! /bin/bash**

**Echo welcome**

**Exit 34**

**Echo bye**

**$ ./welcome.sh**

**#! /bin/bash**

**Echo welcome**

**Exit 34**

**Echo $?**

**Echo bye**

**$ ./welcome.sh**

**$ echo $?**

**We can use our own exit codes.**

**#! /bin/bash**

**Echo welcome**

**Exit 34**

**Echo bye**

**$ ./welcome.sh**

**Welcome**

**$ echo $?**

**34**

1. **Script Parameters**

Users often need to pass parameter values to a script, such as a filename, date, etc. Scripts will take different paths or arrive at different values according to the parameters (command arguments) that are passed to them. These values can be text or numbers as in:

**$ ./script.sh /tmp  
$ ./script.sh 100 200**  
  
Within a script, the parameter or an argument is represented with a **$** and a number or special character. The table lists some of these parameters.

|  |  |
| --- | --- |
| **Parameter** | **Meaning** |
| **$0** | Script name |
| **$1** | First parameter |
| **$2**, **$3**, etc. | Second, third parameter, etc. |
| **$\*** | All parameters |
| **$#** | Number of arguments |

**Using Script Parameters**

If you type in the script shown in the figure, make the script executable with **chmod +x param.sh**. Then, run the script giving it several arguments, as shown. The script is processed as follows:

**$0** prints the script name: **param.sh**

**$1** prints the first parameter: **one**

**$2** prints the second parameter: **two**

**$3** prints the third parameter: **three**

**$\*** prints all parameters: **one two three four five**

The final statement becomes: **All done with param.sh**

# Using Script Parameters. A screenshot of the cat param.sh command and its output

# **Using Script Parameters**

1. **Environment Variables**

Most scripts use variables containing a value, which can be used anywhere in the script. These variables can either be user or system-defined. Many applications use such environment variables (already covered in some detail in *Chapter 12: User Environment*) for supplying inputs, validation, and controlling behaviour.

As we discussed earlier, some examples of standard environment variables are **HOME**, **PATH**, and **HOST**. When referenced, environment variables must be prefixed with the **$** symbol, as in **$HOME**. You can view and set the value of environment variables. For example, the following command displays the value stored in the **PATH** variable:

**$ echo $PATH**

However, no prefix is required when setting or modifying the variable value. For example, the following command sets the value of the **MYCOLOR** variable to blue:

**$ MYCOLOR=blue**

You can get a list of environment variables with the **env**, **set**, or **printenv** commands.

# Environment Variables: a screenshot with different environment variables: echo $MY_FAVORITE_OS; MY_FAVORITE_OS=Linux; echo $MY_FAVORITE_OS Linux; env | grep LANG

# **Environment Variables**

**Exporting Environment Variables**

While we discussed the exportof environment variables in the section on the "*User Environment*", it is worth reviewing this topic in the context of writing bash scripts.

By default, the variables created within a script are available only to the subsequent steps of that script. Any child processes (sub-shells) do not have automatic access to the values of these variables. To make them available to child processes, they must be promoted to environment variables using the export statement, as in:

**export VAR=value**

or

**VAR=value; export VAR**

While child processes are allowed to modify the value of exported variables, the parent will not see any changes; exported variables are not shared, they are only copied and inherited.

Typing *export* with no arguments will give a list of all currently exported environment variables.

# Exporting Variables: a screenshot of export | head -20

# **Exporting Variables**

**How to export permanently?**

**If you want to store any value at userlevel that should available in any bash of the same user.**

**$ ls -a**

**.bash\_history .bash\_logout .bash\_profile .bashrc .ssh**

**EXPORT Command**

**$ echo $a**

**$ export a =10**

**$ bash**

**$ echo $a**

**$ bash**

**$ echo $a**

**$ bash**

**$ echo $a**

**$ ps**

**In . bashrc file**

**$ vi .bashrc**

**Export a = 50 // add this in second line.**

**$ echo $a**

**$ bash**

**$ echo $a**

**$ source .bashrc**

**Or**

**$ vi .bash\_profile // this file internally calls. bashrc only.**

**// The above 2 files specific to user.**

**$ Ls .bashrc**

**How to make these variables available to System level?**

**Go to configuration files**

**$ Cd /etc**

**// you have to remember this**

**$ cd /etc/profile.d**

**$ ls**

**// create one script file and add all the variables here in this file.**

**Profile.d$ vi test.sh**

**$ sudo vi test.sh**

**$ exit**

**$ echo x**

**# echo X**

**If you export variables in profile.d ,, these files available to all users.**

**// here you can use /etc/profile for this purpose as well but we don’t want to touch this file . Its for cuttom.sh shell script files.**

**/etc/profile.**

**$ vi first.sh**

**#! /bin/bash**

**M=10**

**Export m**

**$ chmod +x first.sh**

**$ ./first.sh**

**$ echo $m**

**blank**

**$ vi first.sh**

**#! /bin/bash**

**M=10**

**Export $$**

**$ echo $m**

**// it will not work normally**

**/ Use source command instead**

**//use source whenever in file we have export command.**

**$ source first.sh**

**$ echo $m**

**$ source first.sh**

**$ uname // it tells what is the OS**

**$ date**

**$ a = uname**

**$ echo $a**

**Uname .**

**// here I wants to print os name not the uname , for this we have to use commnad substitution.**

**$ a=$(uname)**

**$ echo $a**

**$ echo**

**Export: user specific**

1. The export is available to multiple bash under the same user. Once bash exit the data also lost, even if you login with the same user again.

export a=30 // It is available to all multiple bash with same user.

1. Use export in this file. bashrc

$ ls -a

$vi. bashrc

Export a=50

1. Source .bashrc
2. Now you can check with multiple bash processes.
3. If you exit and reconnect the user again the variable not available
4. .bash\_profile also used for the same purpose

**How to set variables to available to entire system?**

Ans: Open a folder /etc/profile.d => write script file inside this folder

/etc/profile.d] **$ sudo vi test.sh**

**Export x=40**

**Export y=50**

**Export z=60**

**Exit and reconnect**

$ echo $x =>40

$ echo $y =>50

$ echo $z =>60

**Note:** If you want to execute without closing the bash process use export.

If you want to use and use you use **/etc/profile. d**

**Note:** you can set variable in **/etc/profile** also but it is not recommended.

1. **Command Substitution**

At times, you may need to substitute the result of a command as a portion of another command. It can be done in two ways:

* + - By enclosing the inner command in **$( )**
    - By enclosing the inner command with backticks **(`)**

The second, backticks form, is deprecated in new scripts and commands. No matter which method is used, the specified command will be executed in a newly launched shell environment, and the standard output of the shell will be inserted where the command substitution is done.

Virtually any command can be executed this way. While both of these methods enable command substitution, the **$()** method allows command nesting. New scripts should always use this more modern method. For example:

**$ ls /lib/modules/$(uname -r)/**

In the above example, the output of the command **uname –r** (which will be something like **5.13.3**), is inserted into the argument for the **ls** command.

# Command Substitution: a screenshot of the commands provided in this section and their output

# **Command Substitution**

**Another way?**

**Use back tick**

**$ a = `date`**

**$echo a**

**// in docker we use backticks**

**$ a = `uname`**

**$echo $a**

**We can use in mathematics**

**$ 1+3-2**

**$ b = $((1+3-2))**

**$ echo $b**

**Exit codes:**

**Basic Syntax and Special Characters**

Scripts require you to follow a standard language syntax. Rules delineate how to define variables and how to construct and format allowed statements, etc. The table lists some special character usages within bash scripts:

**# =>** Used to add a comment, except when used as **\#**, or as **#!** when starting a script

**\ =>** Used at the end of a line to indicate continuation on to the next line

**; =>** Used to interpret what follows as a new command to be executed next

**$ =>** Indicates what follows is an environment variable

* => Redirect output

>> => Append output

< => Redirect input

| => Used to pipe the result into the next command

There are other special characters and character combinations and constructs that scripts understand, such as (..), {..}, [..], &&, ||, ', ", $((...)), some of which we will discuss later.

1. **Splitting Long Commands Over Multiple Lines**

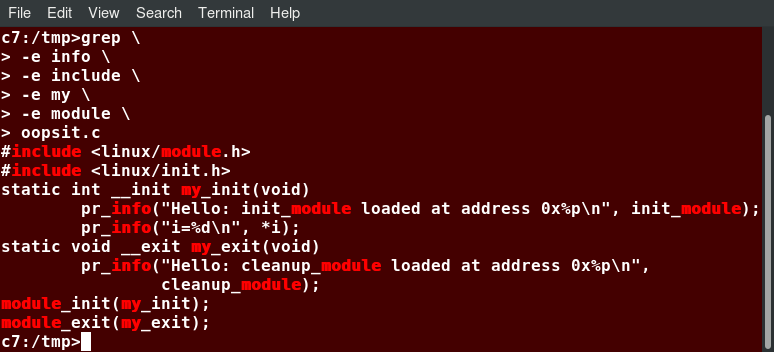
Sometimes, commands are too long to either easily type on one line, or to grasp and understand (even though there is no real practical limit to the length of a command line).

In this case, the concatenation operator (**\**), the backslash character, is used to continue long commands over several lines.

Here is an example of a command installing a long list of packages on a system using Debian package management:

**$~/> cd $HOME  
$~/> sudo apt-get install autoconf automake bison build-essential \chrpath curl diffstat emacs flex gcc-multilib g++-multilib \libsdl1.2-dev libtool lzop make mc patch \ screen socat sudo tar texinfo tofrodos u-boot-tools unzip \ vim wget xterm zip**

The command is divided into multiple lines to make it look readable and easier to understand. The **\** operator at the end of each line causes the shell to combine (concatenate) multiple lines and executes them as one single command.



**Splitting Long Commands Over Multiple Lines**

1. **Putting Multiple Commands on a Single Line**

Users sometimes need to combine several commands and statements and even conditionally execute them based on the behaviour of operators used in between them. This method is called chaining of commands.

There are several different ways to do this, depending on what you want to do. The; (semicolon) character is used to separate these commands and execute them sequentially, as if they had been typed on separate lines. Each ensuing command is executed whether or not the preceding one succeeded.

Thus, the three commands in the following example will all execute, even if the ones preceding them fail:

**$ make ; make install ; make clean**

However, you may want to abort subsequent commands when an earlier one fails. You can do this using the **&&** (and) operator as in:

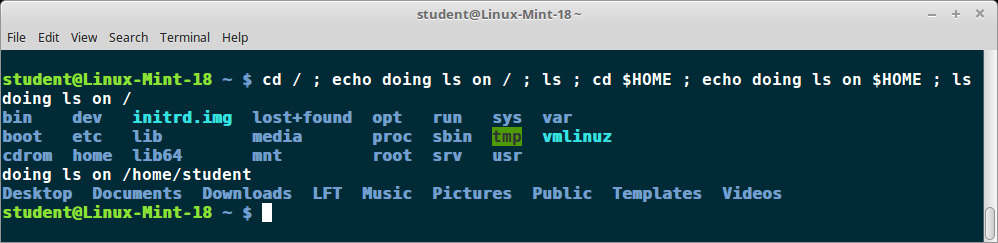
**$ make && make install && make clean**

If the first command fails, the second one will never be executed. A final refinement is to use the **||** (or) operator, as in:

**$ cat file1 || cat file2 || cat file3**

In this case, you proceed until something succeeds and then you stop executing any further steps.

Chaining commands is not the same as piping them; in the later case succeeding commands begin operating on data streams produced by earlier ones before they complete, while in chaining each step exits before the next one starts.



**Putting Multiple Commands on a Single Line**

1. **Input Redirection**

Just as the output can be redirected to a file, the input of a command can be read from a file. The process of reading input from a file is called input redirection and uses the **<** character.

The following three commands (using **wc** to count the number of lines, words and characters in a file) are entirely equivalent and involve input redirection, and a command operating on the contents of a file:

**$ wc < /etc/passwd  
49  105 2678 /etc/passwd**

**$ wc /etc/passwd  
49  105 2678 /etcpasswd**

**$ cat /etc/passwd | wc  
49  105 2678**

1. **Output Redirection**

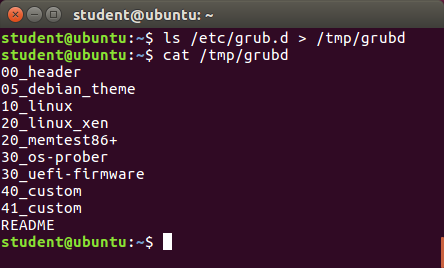
Most operating systems accept input from the keyboard and display the output on the terminal. However, in shell scripting you can send the output to a file. The process of diverting the output to a file is called output redirection. We have already used this facility in our earlier sections on how to use the command line.

The **>** character is used to write output to a file. For example, the following command sends the output of **free** to **/tmp/free.out**:

**$ free > /tmp/free.out**

To check the contents of **/tmp/free.out**, at the command prompt type **cat /tmp/free.out**.

Two **>** characters (**>>**) will append output to a file if it exists, and act just like **>** if the file does not already exist.



**Output Redirection**

1. Conditional Coding
2. Loops
3. Switch case
4. Text Processing

**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**

# The if Statement: a representation of the if statement IF (A=True) Then B Else C End IF

**The if Statement**

# Using the if Statement

In 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.

The elif Statement

You can use the **elif** statement to perform more complicated tests, and take action appropriate actions. The basic syntax is:

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

In the example shown we use strings tests which we will explain shortly, and show how to pull in an environment variable with the **read** statement.

# The elif Statement: a screenshot with an example cat ./show_elif.sh

# **The elif Statement**

# 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:

1. File or directory existence
2. Read or write permission
3. 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. |

If1

#! Bin/bash

Dir = /etc/ssh

If [ -d $dir ] : then

Echo “ssh is available ”

Fi

# chmod +x if.sh

If2

#! Bin/bash

Dir = /etc/ssh

If [ -d $dir ] : then

Echo “ssh is available”

Fi

# pwd

# if – elst

#! Bin/bash

Dir = /etc/ssh

If [ -d $dir ] : then

Echo “ssh available”

Else

Echo “ssh not available”

# Boolean Expressions

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.

# 

# Tests in Boolean Expressions

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**.

# Example of Testing of Strings

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.

|  |  |
| --- | --- |
| **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 |

# Example of Testing of Strings: screenshot of the command cat ./testifstring.sh and its output

# **Example of Testing of Strings**

# 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:

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

# Example of Testing for Numbers

Let us now consider an example of comparing numbers using the various operators:

# Example of Testing for Numbers

# **Example of Testing for Numbers**

But there is so much more that you can do with them.

We are going to be discussing these advanced topics:

manipulating strings, evaluating Boolean expressions, and everybody’s favourite, debugging the scripts you just wrote.

So, let’s begin!

By the end of this chapter, you should be able to:

* + - Manipulate strings to perform actions such as comparison and sorting.
    - Use Boolean expressions when working with multiple data types, including strings or numbers, as well as files.
    - Use **case** statements to handle command line options.
    - Use looping constructs to execute one or more lines of code repetitively.
    - Debug scripts using set **-x** and set **+x**.
    - Create temporary files and directories.
    - Create and use random numbers.

# String Manipulation

Let’s go deeper and find out how to work with strings in scripts.

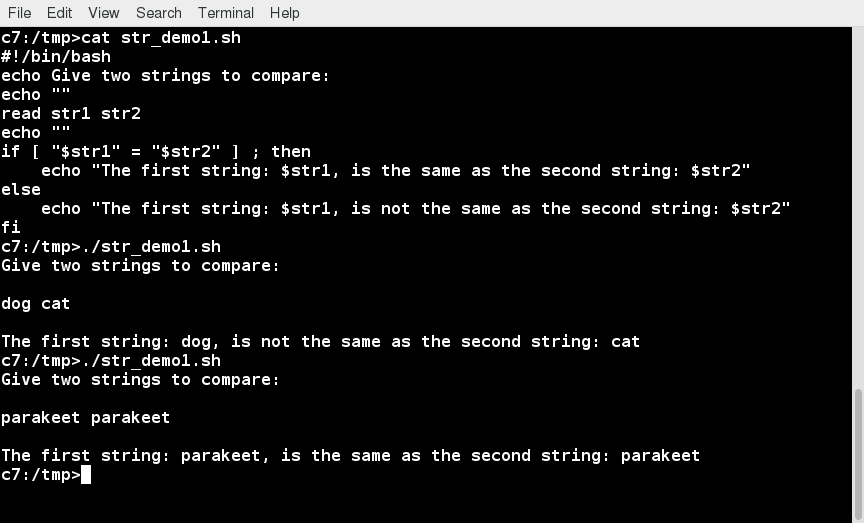
A string variable contains a sequence of text characters. It can include letters, numbers, symbols and punctuation marks. Some examples include: **abcde**, **123**, **abcde 123**, **abcde-123**, **&acbde=%123**.

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**. |

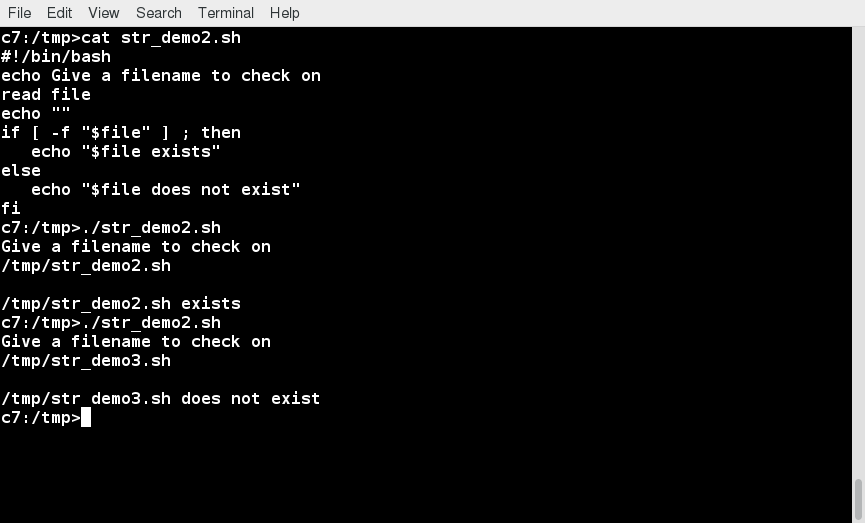
# Example of String Manipulation

In the first example, we compare the first string with the second string and display an appropriate message using the **if** statement.



**Comparing strings and Using if Statement**

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



# Passing a File Name and Checking if It Exists in the Current Directory

# 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#\*.}**.

# Parts of a String - screenshot example

# **Parts of a String**

# Lab 16.1: String Tests and Operations

Write a script which reads two strings as arguments and then:

* + 1. Tests to see if the first string is of zero length, and if the other is of non-zero length, telling the user of both results.
    2. Determines the length of each string, and reports on which one is longer or if they are of equal length.
    3. Compares the strings to see if they are the same, and reports on the result.

Click the link below to view a solution to the Lab exercise.

# Lab Solution: String Tests and Operations

Create a file named **teststrings.sh**, with the content below.

**#!/bin/bash**

**# check two string arguments were given**

**[[ $# -lt 2]] && \**

**echo "Usage: Give two strings as arguments" && exit 1**

**str1=$1**

**str2=$2**

**#------------------------------------**

**## test command**

**echo "Is string 1 zero length? Value of 1 means FALSE"**

**[ -z "$str1" ]**

**echo $?**

**# note if $str1 is empty, the test [ -z $str1 ] would fail**

**# but [[ -z $str1 ]] succeeds**

**# i.e., with [[ ]] it works even without the quotes**

**echo "Is string 2 nonzero length? Value of 0 means TRUE;"**

**[ -n $str2 ]**

**echo $?**

**## comparing the lengths of two string**

**len1=${#str1}**

**len2=${#str2}**

**echo length of string1 = $len1, length of string2 = $len2**

**if [ $len1 -gt $len2 ]**

**then**

**echo "String 1 is longer than string 2"**

**else**

**if [ $len2 -gt $len1 ]**

**then**

**echo "String 2 is longer than string 1"**

**else**

**echo "String 1 is the same length as string 2"**

**fi**

**fi**

**## compare the two strings to see if they are the same**

**if [[ $str1 == $str2 ]]**

**then**

**echo "String 1 is the same as string 2"**

**else**

**if [[ $str1 != $str2 ]]**

**then**

**echo "String 1 is not the same as string 2"**

**fi**

**fi**

**student:/tmp> chmod +x teststrings.sh**

**student:/tmp> ./teststrings.sh str1 str2**

**Is string 1 zero length? Value of 1 means FALSE**

**1**

**Is string 2 nonzero length? Value of 0 means TRUE;**

**0**

**length of string1 = 4, length of string2 = 4**

**String 1 is the same length as string 2**

**String 1 is not the same as string 2**

**student:/tmp>./teststrings.sh str1 str2long**

**Is string 1 zero length? Value of 1 means FALSE**

**1**

**Is string 2 nonzero length? Value of 0 means TRUE;**

**0**

**length of string1 = 4, length of string2 = 8**

**String 2 is longer than string 1**

**String 1 is not the same as string 2**

**student:/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.

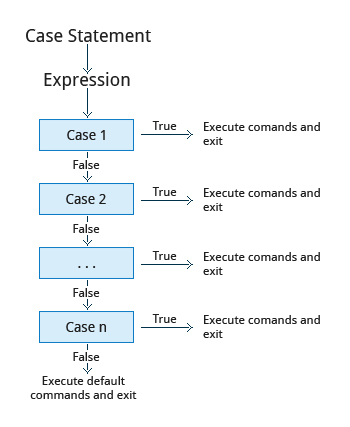
Below are some of the advantages of using the **case** statement:

* + - It is easier to read and write.
    - 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.

# Structure of the case Statement

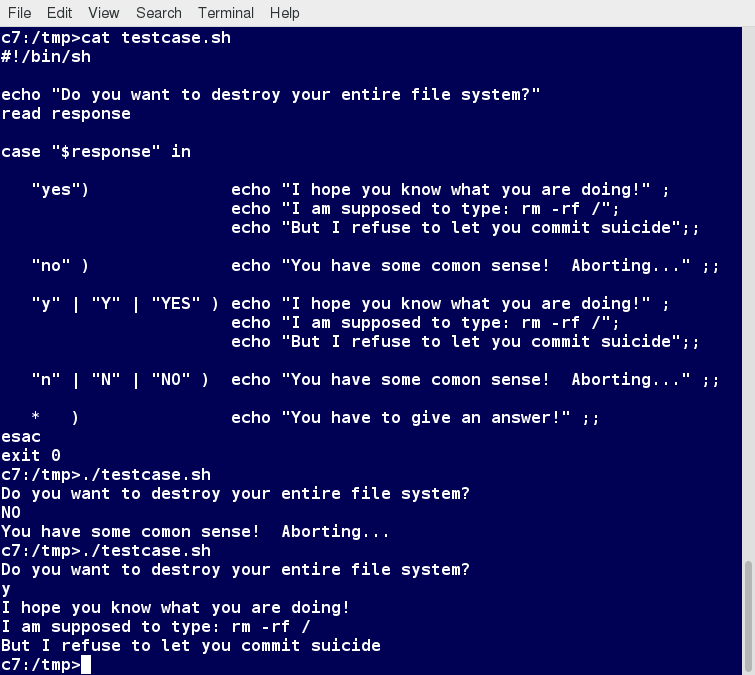
Here is the basic structure of the **case** statement:

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



# 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.



**Example of Use of the case Construct**

# Lab 16.2: Using the case Statement

Write a script that takes as an argument a month in numerical form (i.e. between 1 and 12), and translates this to the month name and displays the result on standard out (the terminal).

If no argument is given, or a bad number is given, the script should report the error and exit.

Click the link below to view a solution to the Lab exercise.

[**Lab Solution**](https://courses.edx.org/asset-v1:LinuxFoundationX+LFS101x+2T2021+type@asset+block/labsol-case.html)

# Lab Solution: Using the case Statement

Create a file named **testcase.sh**, with the content below.

**#!/bin/bash**

**# Accept a number between 1 and 12 as**

**# an argument to this script, then return the**

**# the name of the month that corresponds to that number.**

**# Check to see if the user passed a parameter.**

**if [ $# -eq 0 ]**

**then**

**echo "Error. Give as an argument a number between 1 and 12."**

**exit 1**

**fi**

**# set month equal to argument passed for use in the script**

**month=$1**

**################################################**

**# The example of a case statement:**

**case $month in**

**1) echo "January" ;;**

**2) echo "February" ;;**

**3) echo "March" ;;**

**4) echo "April" ;;**

**5) echo "May" ;;**

**6) echo "June" ;;**

**7) echo "July" ;;**

**8) echo "August" ;;**

**9) echo "September" ;;**

**10) echo "October" ;;**

**11) echo "November" ;;**

**12) echo "December" ;;**

**\*)**

**echo "Error. No month matches: $month"**

**echo "Please pass a number between 1 and 12."**

**exit 2**

**;;**

**esac**

**exit 0**

Make it executable and run it:

**student:/tmp> chmod +x testcase.sh**

**student:/tmp> ./testcase.sh 5**

**May**

**student:/tmp> ./testcase.sh 12**

**December**

**student:/tmp> ./testcase.sh 99**

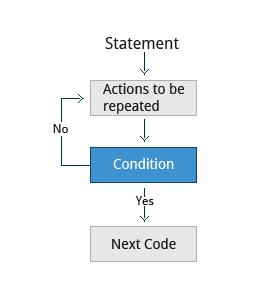
**Error. No month matches: 99**

**Please pass a number between 1 and 12**

**student:/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 return either true or false, as is required.



Three types 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.

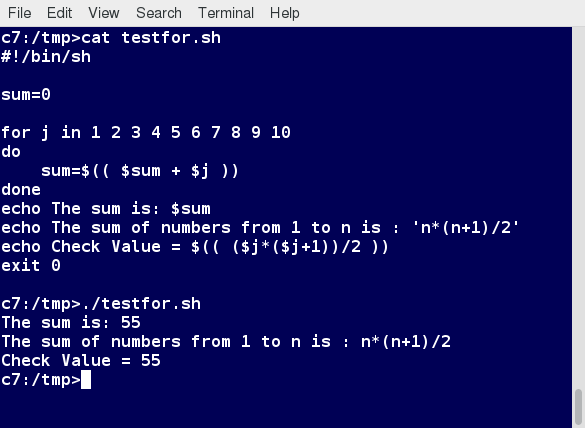
# The for Loop

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.



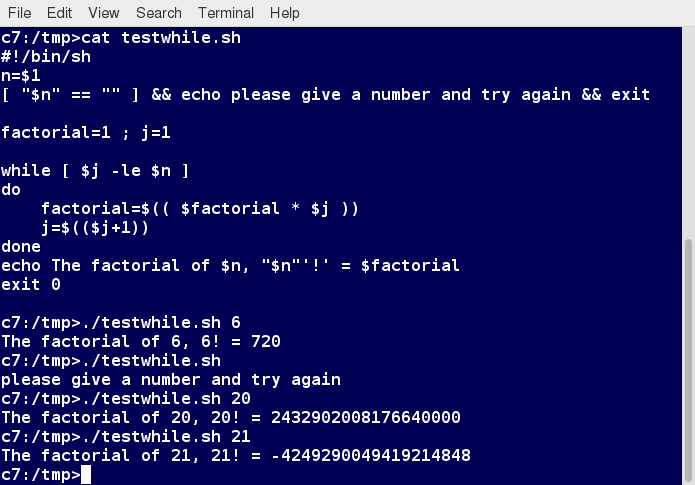
# The while Loop

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?



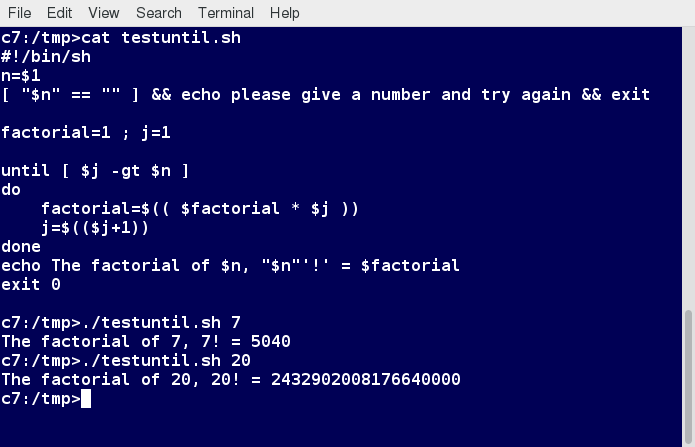
# The until Loop

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 an 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

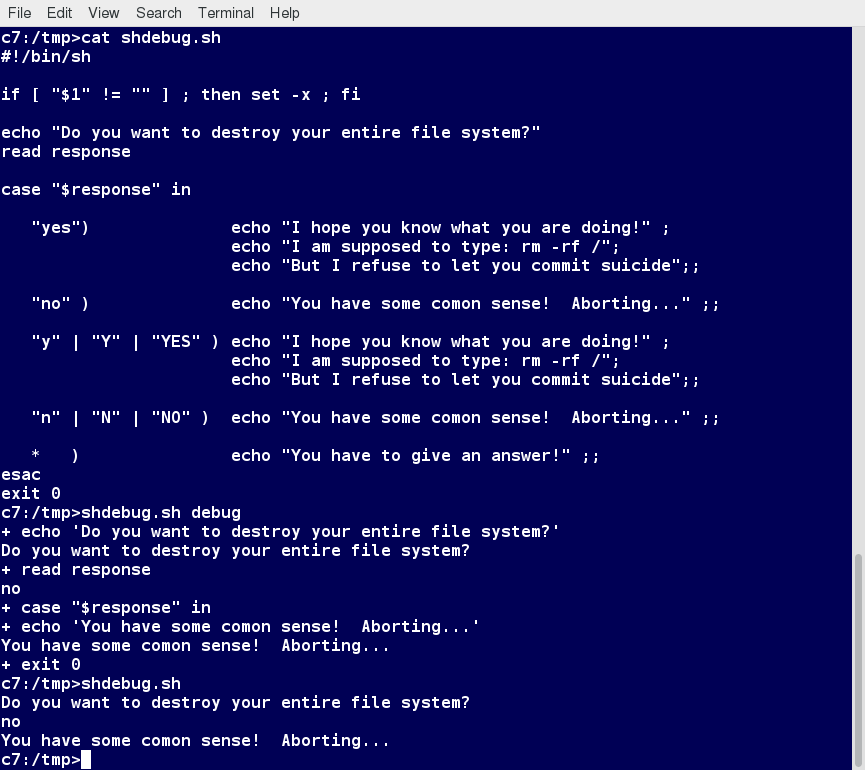
Before fixing an error (or bug), it is vital to know its source.

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

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?

# Redirecting Errors to File and Screen

# **Redirecting Errors to File and Screen**

# 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 |

# 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.

# Example of Creating a Temporary File and Directory

# **Example of Creating a Temporary File and Directory**

# 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**.

# Discarding Output with /dev/null - screenshot

# **Discarding Output with /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"*](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.

# Random Numbers and Data: screenshot: for n in 1 2 3 4 5 do echo A New Random Number is $RANDOM done

# **Random Numbers and Data**

# 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.

# How the Kernel Generates Random Numbers: screenshot of ls -l /dev/*random

# **How the Kernel Generates Random Numbers**

# Lab 16.3: Using Random Numbers

Write a script which:

* + 1. Takes a word as an argument.
    2. Appends a random number to it.
    3. Displays the answer.

Click the link below to view a solution to the Lab exercise.

# Lab Solution: Using Random Numbers

Create a file named **testrandom.sh**, with the content below.

**#!/bin/bash**

**##**

**# check to see if the user supplied in the parameter.**

**[[ $# -eq 0 ]] && echo "Usage: $0 word" && exit 1**

**echo "$1-$RANDOM"**

**exit 0**

Make it executable and run it:

**student:/tmp> chmod +x testrandom.sh**

**student:/tmp> ./testrandom.sh strA**

**strA-29294**

**student:/tmp>./testrandom.sh strB**

**strB-23911**

**student:/tmp>./testrandom.sh strC**

**strC-27782**

**student:/tmp>**