Bash Shell Scripting String Manipulation

String Manipulation:

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

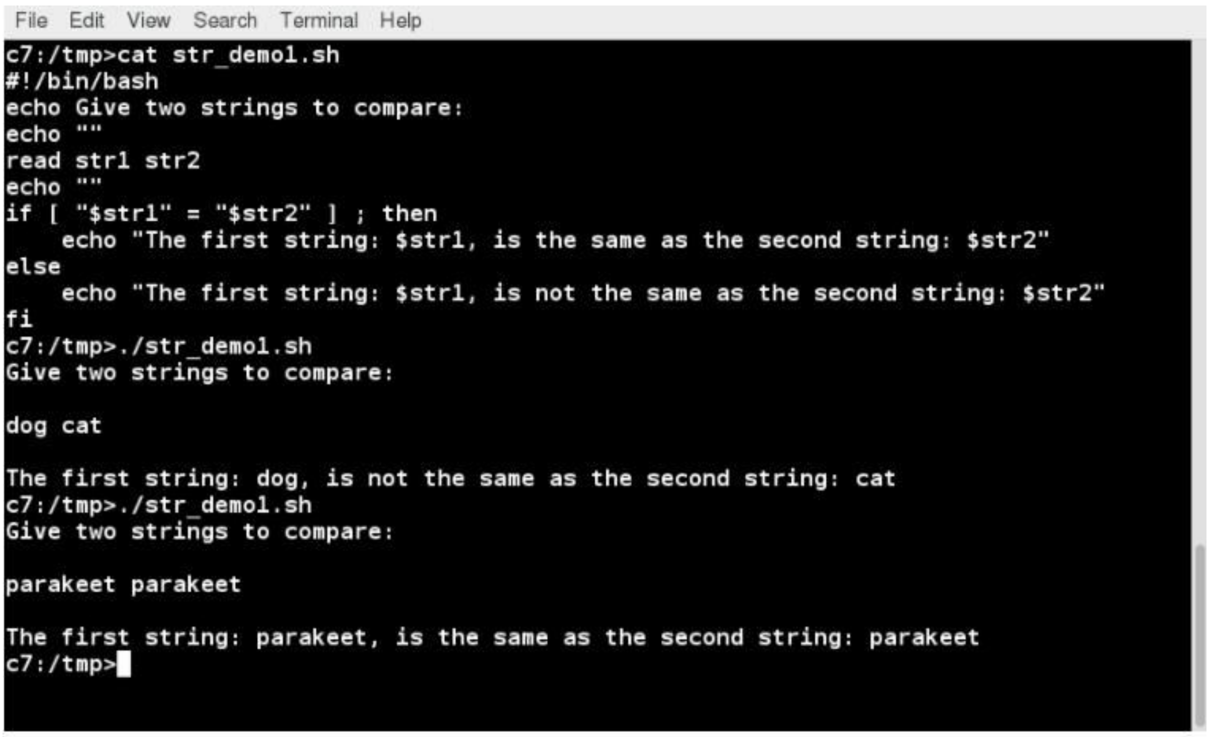
A string variable contains a sequence of text characters. It can include letters, numbers, symbols and punctuation marks. Some examples include: **`abcde`**, **`123`**, **`abcde 123`**, **`abcde-123`**, **`&acbde=%123`**.

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:

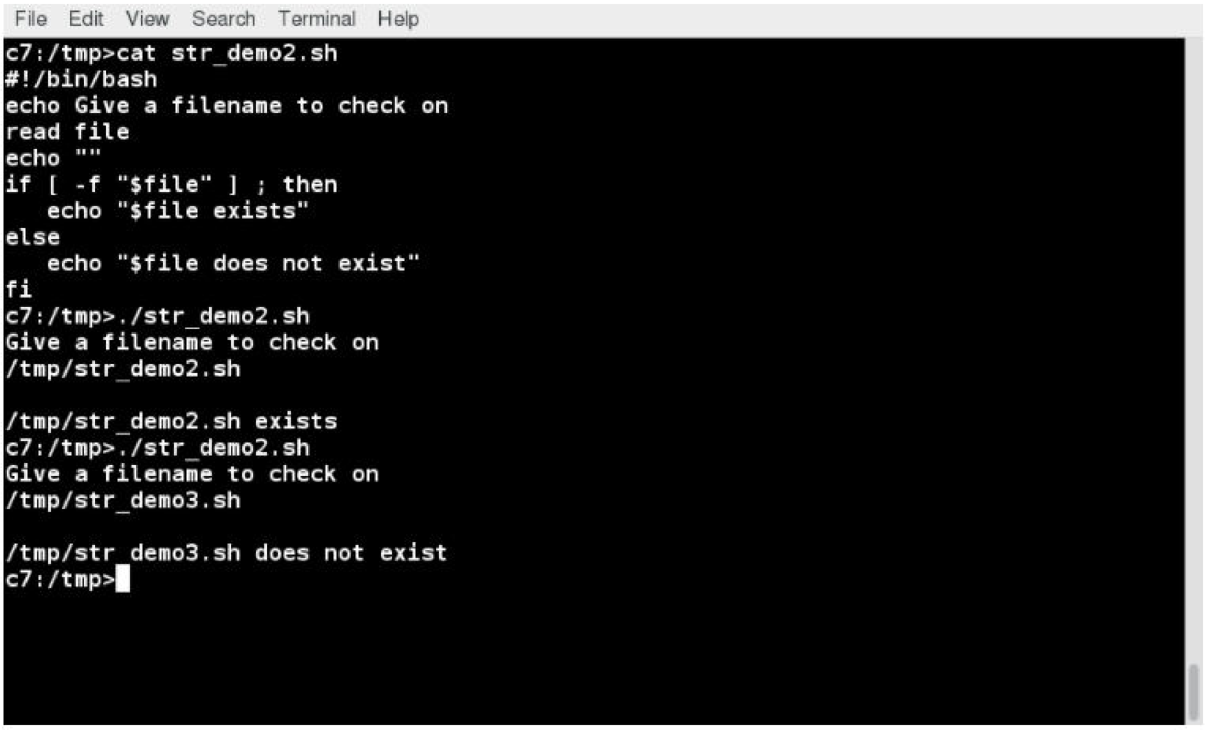
In the first example, we compare the first string with the second string and display an appropriate message using the `if` statement.



```
c7:/tmp>cat str_demo1.sh
#!/bin/bash
echo Give two strings to compare:
echo ""
read str1 str2
echo ""
if [ "$str1" = "$str2" ] ; then
    echo "The first string: $str1, is the same as the second string: $str2"
else
    echo "The first string: $str1, is not the same as the second string: $str2"
fi
c7:/tmp>./str_demo1.sh
Give two strings to compare:

dog cat

The first string: dog, is not the same as the second string: cat
c7:/tmp>./str_demo1.sh
Give two strings to compare:

parakeet parakeet

The first string: parakeet, is the same as the second string: parakeet
c7:/tmp>
```

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.



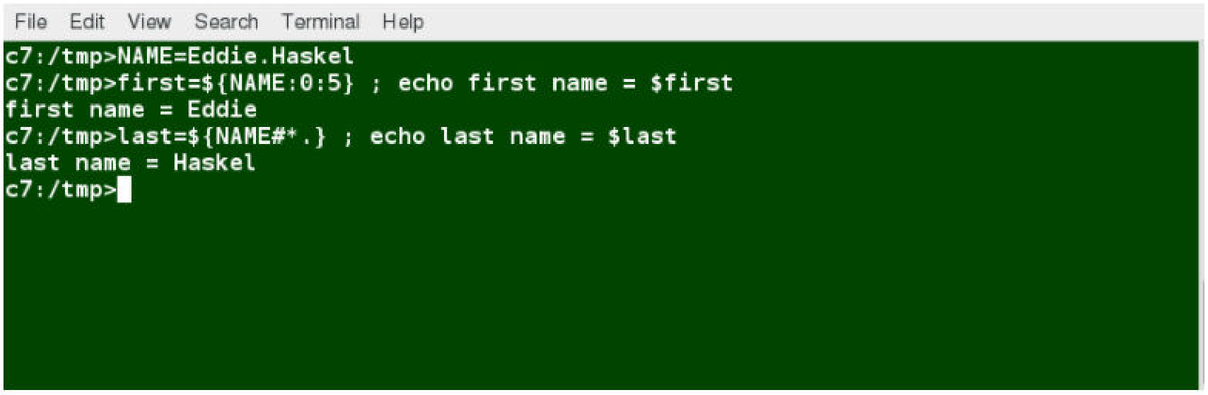
```
c7:/tmp>cat str_demo2.sh
#!/bin/bash
echo Give a filename to check on
read file
echo ""
if [ -f "$file" ] ; then
   echo "$file exists"
else
   echo "$file does not exist"
fi
c7:/tmp>./str_demo2.sh
Give a filename to check on
/tmp/str_demo2.sh

/tmp/str_demo2.sh exists
c7:/tmp>./str_demo2.sh
Give a filename to check on
/tmp/str_demo3.sh

/tmp/str_demo3.sh does not exist
c7:/tmp>
```

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#*.}**.



```
c7:/tmp>NAME=Eddie.Haskel
c7:/tmp>first=${NAME:0:5} ; echo first name = $first
first name = Eddie
c7:/tmp>last=${NAME#*.} ; echo last name = $last
last name = Haskel
c7:/tmp>
```

Parts of a String

Lab 16.1: String Tests and Operations

Bookmark this page

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.

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
```

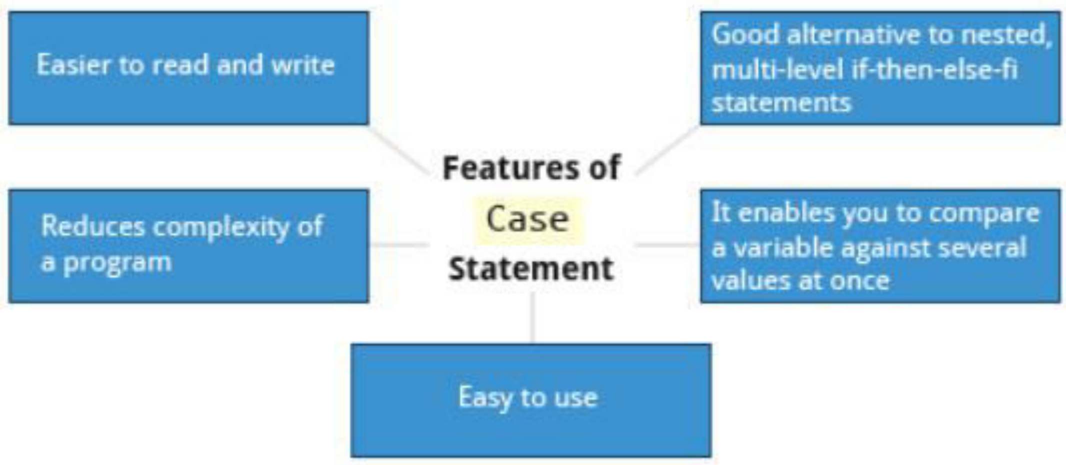
```
student:/tmp> chmod +x teststrings.sh
student:/tmp> ./teststrings.sh str1 str2


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.

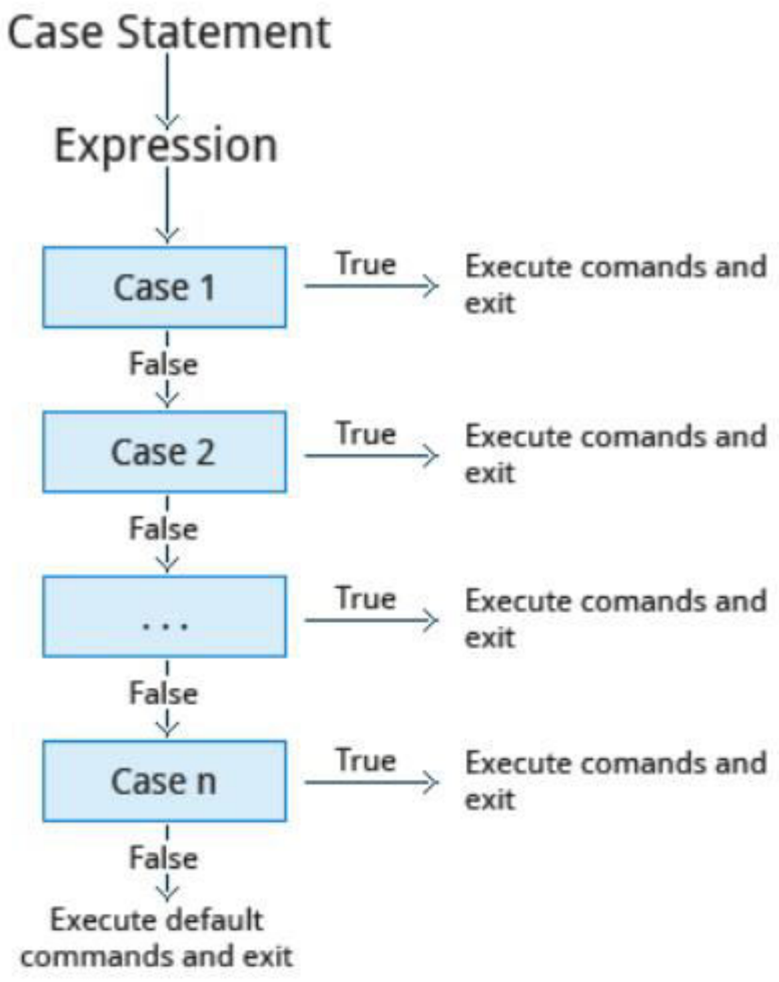

Easier to read and write
Good alternative to nested, multi-level if-then-else-fi statements
Features of Case Statement
Reduces complexity of a program
It enables you to compare a variable against several values at once
Easy to use

Features of case Statement

Structure of the case Statement:

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

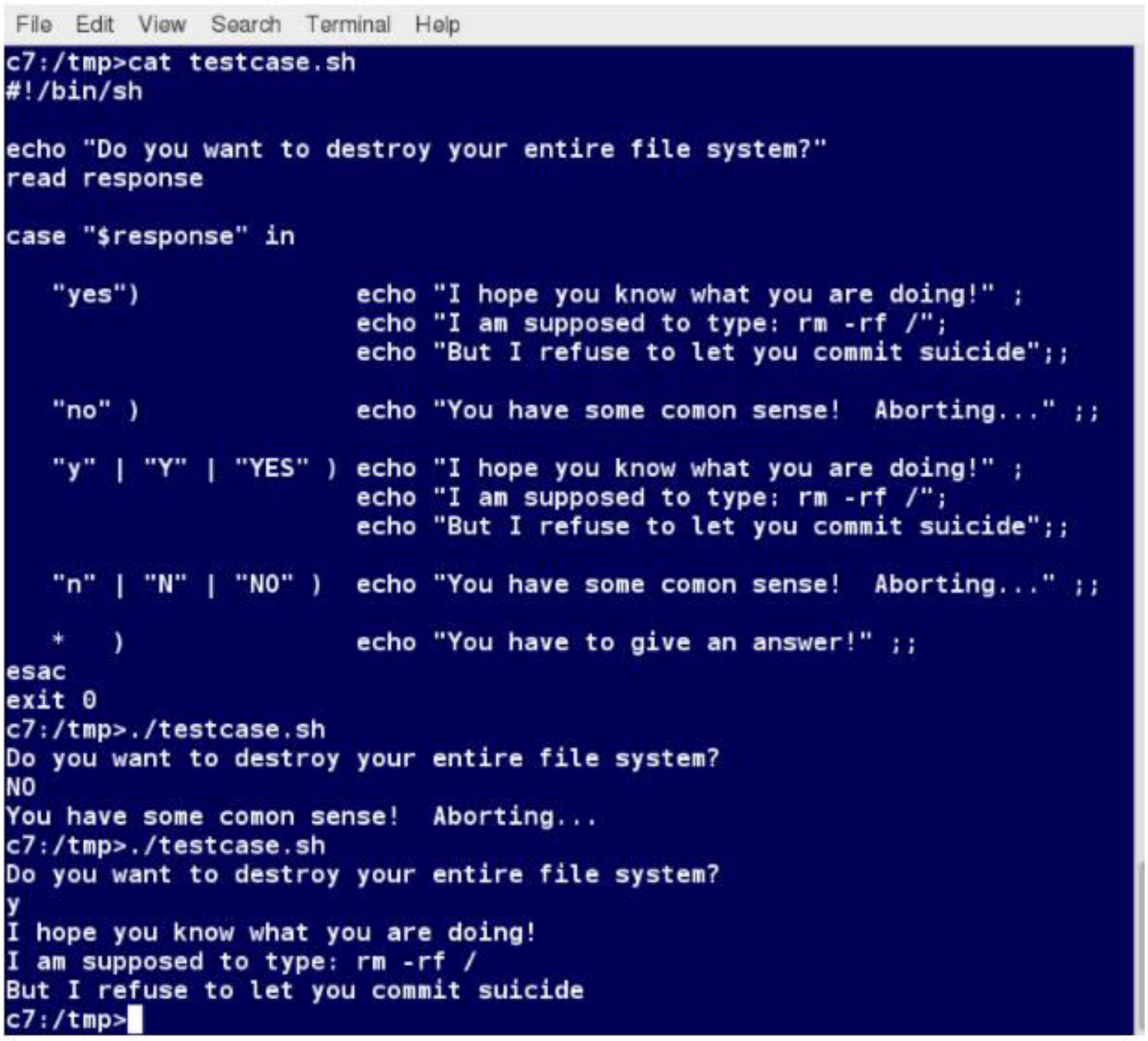
```
case expression in
   pattern1) execute commands;;
   pattern2) execute commands;;
   pattern3) execute commands;;
   pattern4) execute commands;;
   * )       execute some default commands or nothing ;;
esac
```



Structure of the `case` Statement

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.



```
c7:/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!" ;
                      echo "I am supposed to type: rm -rf /";
                      echo "But I refuse to let you commit suicide";;

   "no" )             echo "You have some comon sense!  Aborting..." ;;

   "y" | "Y" | "YES" ) echo "I hope you know what you are doing!" ;
                      echo "I am supposed to type: rm -rf /";
                      echo "But I refuse to let you commit suicide";;

   "n" | "N" | "NO" )  echo "You have some comon sense!  Aborting..." ;;

   *   )              echo "You have to give an answer!" ;;
esac
exit 0
c7:/tmp>./testcase.sh
Do you want to destroy your entire file system?
NO
You have some comon sense!  Aborting...
c7:/tmp>./testcase.sh
Do you want to destroy your entire file system?
y
I hope you know what you are doing!
I am supposed to type: rm -rf /
But I refuse to let you commit suicide
c7:/tmp>
```

Example of Use of the `case` Construct

Lab 16.2: Using the case Statement

Bookmark this page

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.

Lab Solution: Using the case Statement

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

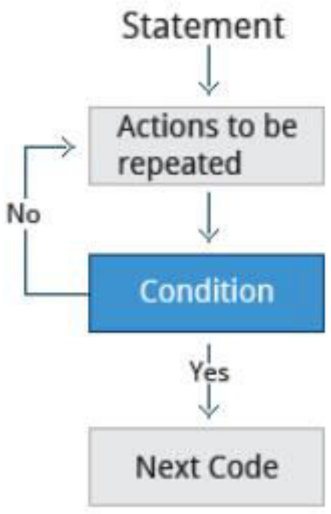
```
#!/bin/bash

# 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 returns either true or false, as is required.



Looping Constructs

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.

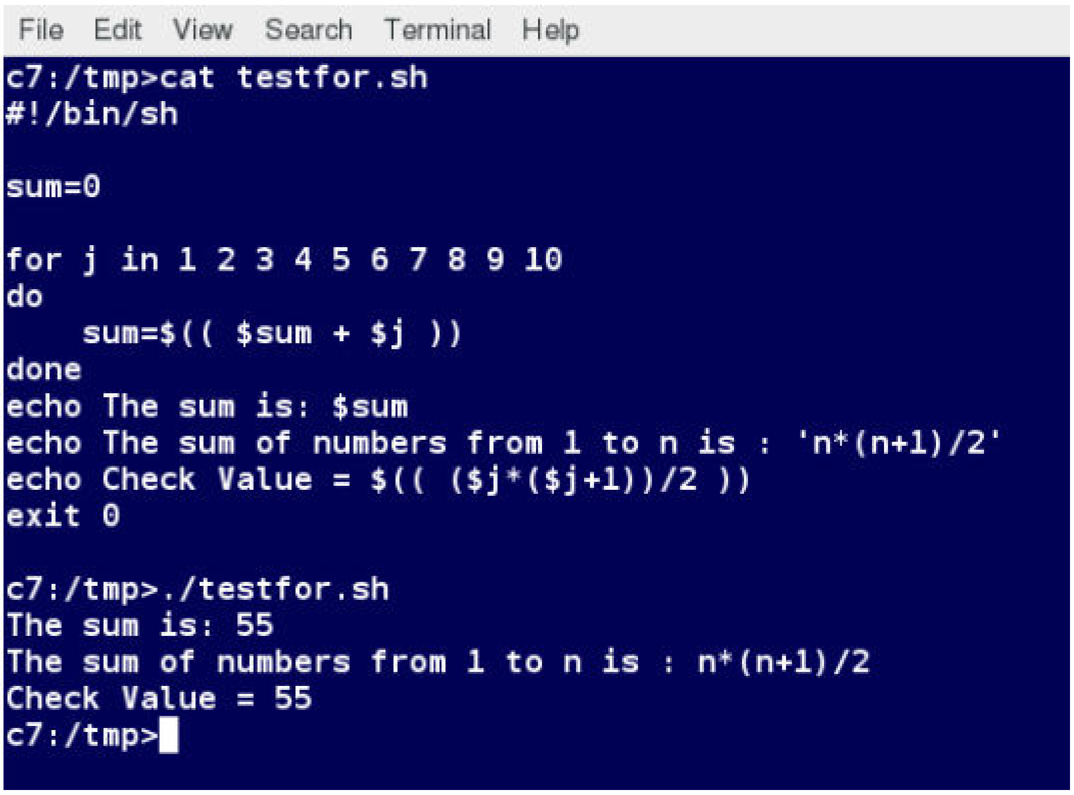
The for Loop:

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.



```
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>
```

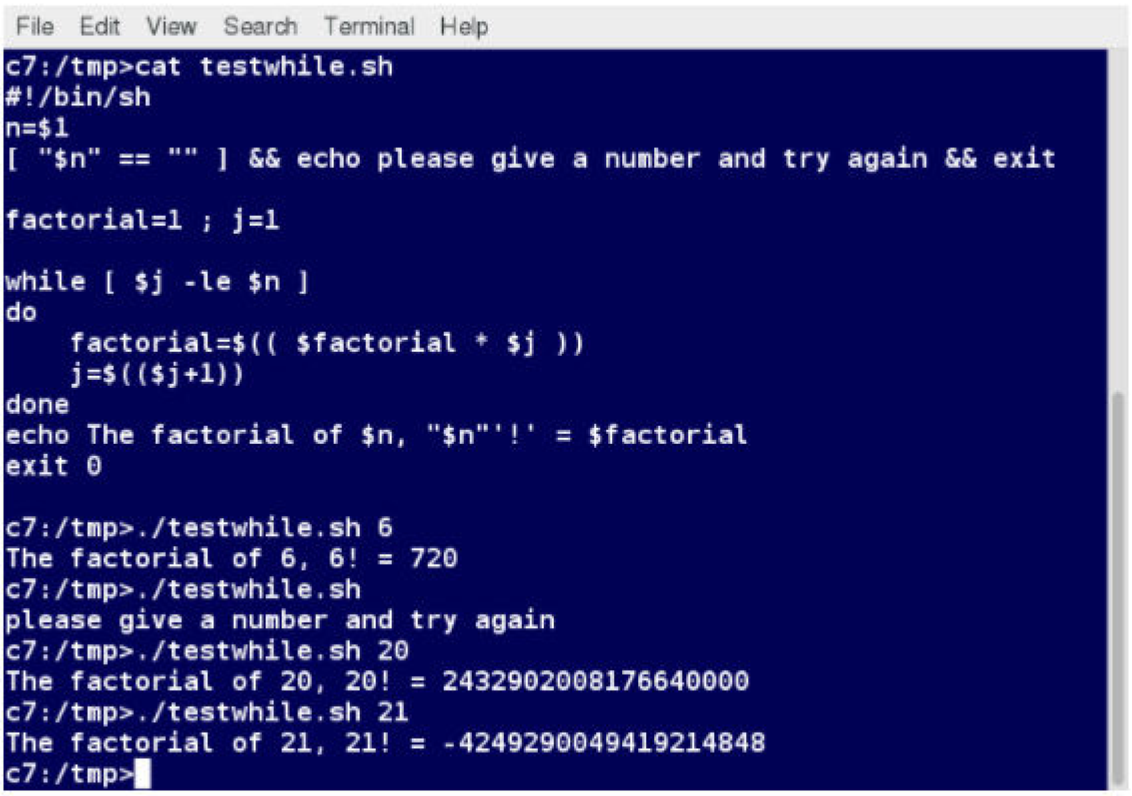
The while Loop:

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?



```
c7:/tmp>cat testwhile.sh
#!/bin/sh
n=$1
[ "$n" == "" ] && echo please give a number and try again && exit

factorial=1 ; j=1

while [ $j -le $n ]
do
    factorial=$(( $factorial * $j ))
    j=$(($j+1))
done
echo The factorial of $n, "$n"'!' = $factorial
exit 0

c7:/tmp>./testwhile.sh 6
The factorial of 6, 6! = 720
c7:/tmp>./testwhile.sh
please give a number and try again
c7:/tmp>./testwhile.sh 20
The factorial of 20, 20! = 2432902008176640000
c7:/tmp>./testwhile.sh 21
The factorial of 21, 21! = -4249290049419214848
c7:/tmp>
```

The **while** Loop

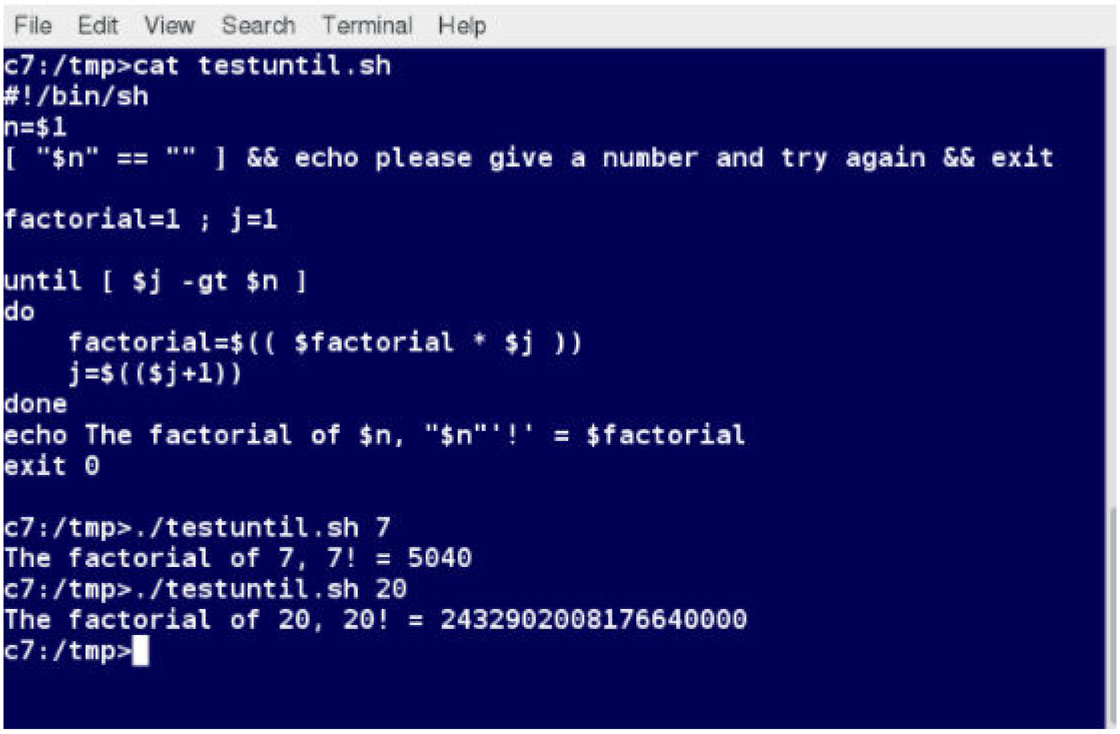
The until Loop:

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.



```
c7:/tmp>cat testuntil.sh
#!/bin/sh
n=$1
[ "$n" == "" ] && echo please give a number and try again && exit

factorial=1 ; j=1

until [ $j -gt $n ]
do
    factorial=$(( $factorial * $j ))
    j=$(($j+1))
done
echo The factorial of $n, "$n"'!' = $factorial
exit 0

c7:/tmp>./testuntil.sh 7
The factorial of 7, 7! = 5040
c7:/tmp>./testuntil.sh 20
The factorial of 20, 20! = 2432902008176640000
c7:/tmp>
```

The `until` 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:

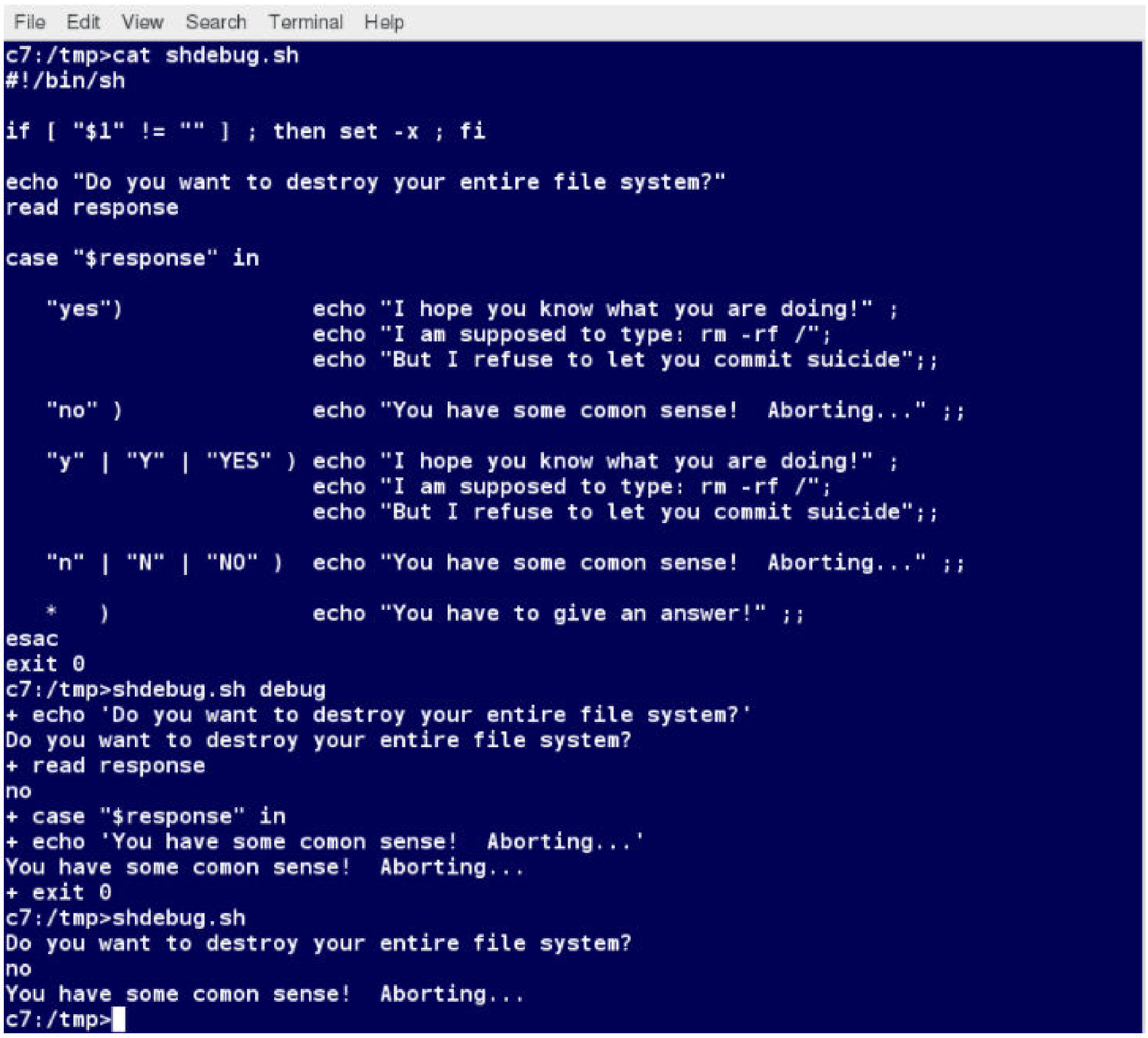
Before fixing an error (or bug), it is vital to know its source.

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.



```
c7:/tmp>cat shdebug.sh
#!/bin/sh

if [ "$1" != "" ] ; then set -x ; fi

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!" ;
                        echo "I am supposed to type: rm -rf /";
                        echo "But I refuse to let you commit suicide";;

   "no" )               echo "You have some comon sense!  Aborting..." ;;

   "y" | "Y" | "YES" )  echo "I hope you know what you are doing!" ;
                        echo "I am supposed to type: rm -rf /";
                        echo "But I refuse to let you commit suicide";;

   "n" | "N" | "NO" )   echo "You have some comon sense!  Aborting..." ;;

   *   )                echo "You have to give an answer!" ;;
esac
exit 0
c7:/tmp>shdebug.sh debug
+ echo 'Do you want to destroy your entire file system?'
Do you want to destroy your entire file system?
+ read response
no
+ case "$response" in
+ echo 'You have some comon sense!  Aborting...'
You have some comon sense!  Aborting...
+ exit 0
c7:/tmp>shdebug.sh
Do you want to destroy your entire file system?
no
You have some comon sense!  Aborting...
c7:/tmp>
```

Script Debug Mode

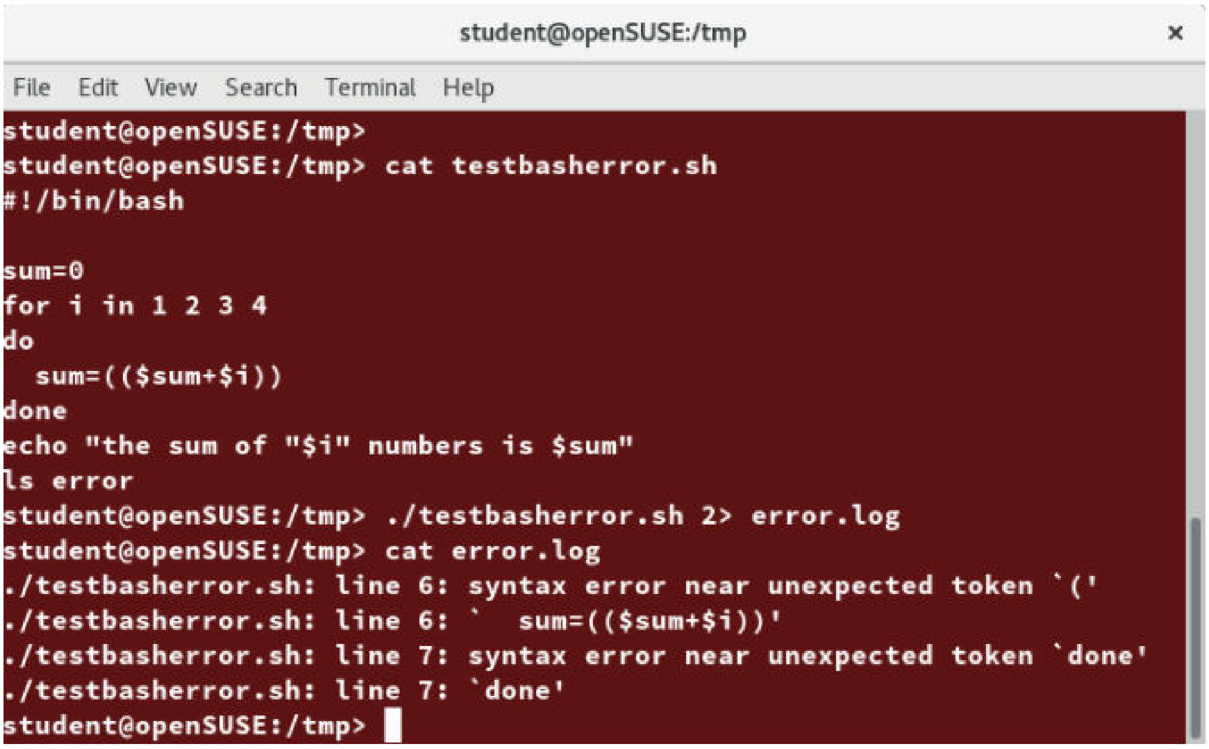
Redirecting Errors to File and Screen:

1. 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?



```
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> 
```

Redirecting Errors to File and Screen