MARUDHAR KESARI JAIN COLLEGE FOR WOMEN, VANIYAMBADI
PG DEPARTMENT OF COMPUTER APPLICATIONS

Subject code : 23PCA12

Class : I-MCA

SUBJECT: LINUX AND SHELL PROGRAMMING

Linux and shell programming

UNIT-III

Creating functions: Basic script functions returning a value-Using variable in Function –Arrays and variable functions-Functions recursion-Creating a library-Using function on the command line-writing script for graphical Desktop: Creating text menus-Building text windows widgets adding x windows graphics .Introducing sed and gawk: Learning about theseteditorbasics.

Basic Function:

The bash shell provides a feature allowing you to do just that. Functions are blocks of script codethat you assign a name to and reuse anywhere in your code. Anytime you need to use that block of code in your script, you simply use the function name you assigned it (referred to as calling the function). This section describes how to create and use functions in your shell scripts.

function name
{
commands
}

The name attribute defines a unique name assigned to the function. Each function youdefine in your script must be assigned a unique name.

The commands are one or more bash shell commands that make up your function. Whenyou call the function, the bash shell executes each of the commands in the order theyappear in the function, just as in a normal script.The second format for defining a function in a bash shell script more closely follows howfunctions are defined in other programming languages:

name()

{

commands

}

The empty parentheses after the function name indicate that you're defining a function.The same naming rules apply in this format as in the original shell script function format.

Using functions

To use a function in your script, specify the function name on a line, just as you would any

other shell command:

```
$ cat test1
#!/bin/bash
# using a function in a script
function func1 {
echo "This is an example of a function"
}
count=1
while [ $count -le 5 ]
do
func1
count=$[ $count + 1 ]
done
echo "This is the end of the loop"
func1
echo "Now this is the end of the script"
$
$ ./test1
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
```

This is an example of a function
This is the end of the loop
This is an example of a function
Now this is the end of the script

Each time you reference the func1 function name, the bash shell returns to thefunc1 function definition and executes any commands you defined there.

The function definition doesn't have to be the fi rst thing in your shell script, but be care-
ful. If you attempt to use a function before it'sdefined, you'll get an error message:

```
$ cat test2
#!/bin/bash
# using a function located in the middle of a script
count=1
echo "This line comes before the function definition"
function func1 {
echo "This is an example of a function"
}
while [ $count -le 5 ]
do
func1
count=$[ $count + 1 ]
done
echo "This is the end of the loop"
func2
echo "Now this is the end of the script"
function func2 {
```

```
echo "This is an example of a function"
}
$
$ ./test2
This line comes before the function definition
This is an example of a function
```



```
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is the end of the loop
./test2: func2: command not found
Now this is the end of the script
$
```

The first function, func1, was defined after a couple of statements in the script, which isperfectly fine. When the func1 function was used in the script, the shell knew where tofind it.

Returning Value:

The bash shell treats functions like mini-scripts, complete with an exit status .There are three different ways you can generate an exit status for your functions.

The default exit statusof a function is the exit status returned by the last command inthe function. After the function executes, you use the standard $? variable to determine

the exit status of the function:

```
$ cat test4
#!/bin/bash
# testing the exit status of a function
func1() {
echo "trying to display a non-existent file"
ls -l badfile
}
echo "testing the function: "
func1
echo "The exit status is: $?"
$
$ ./test4
testing the function:
trying to display a non-existent file
ls: badfile: No such file or directory
The exit status is: 1
$
```

The exit status of the function is 1 because the last command in the function failed.

However, you have no way of knowing if any of the other commands in the function com-pleted successfully or not. Look at this example:

```
$ cat test4b
#!/bin/bash
# testing the exit status of a function
func1() {
ls -l badfile
echo "This was a test of a bad command"
}
```

```
echo "testing the function:"
func1
echo "The exit status is: $?"
$
$ ./test4b
testing the function:
ls: badfile: No such file or directory
This was a test of a bad command
The exit status is: 0
$
```

This time, because the function ended with an echo statement that completed successfully,the exit status of the function is 0, even though one of the commands in the function failed. Using the default exit status of a function can be a dangerous practice.

Using the return command:

Using the return command

The bash shell uses the return command to exit a function with a specifi c exit status.The return command allows you to specify a single integer value to defi ne the functionexit status, providing an easy way for you to programmatically set the exit status of yourfunction:

```
$ cat test5
#!/bin/bash
# using the return command in a function
functiondbl {
read -p "Enter a value: " value
echo "doubling the value"
return $[ $value * 2 ]
}
dbl
```

```
echo "The new value is $?"
$
```

The dbl function doubles the integer value contained in the $value variable provided by the user input. It then returns the result using the return command, which the script dis-plays using the $? variable.

Keep the following two tips in mind to avoid problems:

■ Remember to retrieve the return value as soon as the function completes.

■ Remember that an exit status must be in the range of 0 to 255.

The second problem defines a limitation for using this return value technique. Because anexit status must be less than 256, the result of your function must produce an integer value less than 256. Any value over that returns an error value:

```
$ ./test5
Enter a value: 200
doubling the value
The new value is 1
```

Using function output

Just as you can capture the output of a command to a shell variable, you can also capture

the output of a function to a shell variable. You can use this technique to retrieve any type

of output from a function to assign to a variable:

```
result='dbl'
```

This command assigns the output of the dbl function to the $result shell variable. Here’s

an example of using this method in a script:

```
$ cat test5b
#!/bin/bash
```

```
# using the echo to return a value
functiondbl {
read -p "Enter a value: " value
echo $[ $value * 2 ]
}
result=$(dbl)
echo "The new value is $result"
$
$ ./test5b
Enter a value: 200
The new value is 400
$
$ ./test5b
456
```



```
Enter a value: 1000
The new value is 2000
$
```

The new function now uses an echo statement to display the result of the calculation. The script just captures the output of the dbl function instead of looking at the exit status for the answer.

USING THE VARIABLE IN FUNCTION:

we used a variable called $value within the function to hold the value that it processed. When you use variables in your functions, you need to be somewhat careful about how you defi ne and handle them. This is a common cause of problems in shell scripts. This section goes over a few techniques for handling variables both inside and outside your shell script functions.

Passing parameters to a function

As mentioned earlier in the “Returning a Value” section, the bash shell treats functionsjust like mini-scripts. This means that you can pass parameters to a function just like a regular script

Functions can use the standard parameter environment variables to represent any parameters passed to the function on the command line. For example, the name of the function isdefined in the $0 variable, and any parameters on the function command line are define using the variables $1, $2, and so on.

func1 $value1 10

The function can then retrieve the parameter values using the parameter environment

variables. Here’s an example of using this method to pass values to a function:

```
$ cat test6
#!/bin/bash
# passing parameters to a function
functionaddem {
if [ $# -eq 0 ] || [ $# -gt 2 ]
then
echo -1
elif [ $# -eq 1 ]
then
echo $[ $1 + $1 ]
else
echo $[ $1 + $2 ]
fi
}
echo -n "Adding 10 and 15: "
```

```
value=$(addem 10 15)
echo $value
echo -n "Let's try adding just one number: "
value=$(addem 10)
echo $value
echo -n "Now trying adding no numbers: "
value=$(addem)
echo $value
echo -n "Finally, try adding three numbers: "
value=$(addem 10 15 20)
echo $value
$
$ ./test6
Adding 10 and 15: 25
Let's try adding just one number: 20
Now trying adding no numbers: -1
Finally, try adding three numbers: -1
$
```

The addem function in the text6 script fi rst checks the number of parameters passed toit by the script. If there aren't any parameters, or if there are more than two parameters,addem returns a value of -1. If there's just one parameter, addem adds the parameter toitself for the result. If there are two parameters, addem adds them together for the result.

```
$ cat badtest1
#!/bin/bash
# trying to access script parameters inside a function
function badfunc1 {
echo $[ $1 * $2 ]
}
```

```
if [ $# -eq 2 ]
then
value=$(badfunc1)
echo "The result is $value"
else
echo "Usage: badtest1 a b"
fi
$
$ ./badtest1
Usage: badtest1 a b
$ ./badtest1 10 15
./badtest1: * : syntax error: operand expected (error token is "*
")
The result is
$
```

Even though the function uses the $1 and $2 variables, they aren’t the same $1 and $2

Even though the function uses the $1 and $2 variables, they aren’t the same $1 and $2

variables available in the main part of the script. Instead, if you want to use those values

in your function, you have to manually pass them when you call the function:

```
$ cat test7
#!/bin/bash
# trying to access script parameters inside a function
function func7 {
echo $[ $1 * $2 ]
}
```

```
if [ $# -eq 2 ]
then
value=$(func7 $1 $2)
echo "The result is $value"
else
echo "Usage: badtest1 a b"
fi
$
$ ./test7
Usage: badtest1 a b
$ ./test7 10 15
The result is 150
$
```

By passing the $1 and $2 variables to the function, they become available for the function to use, just like any other parameter.

Array Variables and Functions

advanced the way of allowing a single variable to hold multiple values sing arrays. Using array variable values with functions is a little tricky, and there aresome special considerations. This section describes a technique that allows you to do that.

Passing arrays to functions

The art of passing an array variable to a script function can be confusing. If you try to pass
the array variable as a single parameter, it doesn't work

```
$ cat badtest3
#!/bin/bash
```

the array variable into its individual values
and use the values as function parameters. Inside the function, you can reassemble all the
parameters into a new array variable. Here's an example of doing this:

```
$ cat test10
#!/bin/bash
```

```
# array variable to function test
functiontestit {
localnewarray
newarray=(;'echo "$@"')
echo "The new array value is: ${newarray[*]}"
}
myarray=(1 2 3 4 5)
echo "The original array is ${myarray[*]}"
testit ${myarray[*]}
$
$ ./test10
The original array is 1 2 3 4 5
The new array value is: 1 2 3 4 5
$
```

The script uses the $myarray variable to hold all the individual array values to place them all on the command line for the function. The function then rebuilds the array variable from the command line parameters. Once inside the function, the array can be used just like any other array:

```
$ cat test11
#!/bin/bash
# adding values in an array
functionaddarray {
local sum=0
localnewarray
newarray=($(echo "$@"))
for value in ${newarray[*]}
do
sum=$[ $sum + $value ]
done
echo $sum
}
myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
arg1=$(echo ${myarray[*]})
result=$(addarray $arg1)
echo "The result is $result"
$
$ ./test11
The original array is: 1 2 3 4 5
The result is 15
$
```

The addarray function iterates through the array values, adding them together. You can
put any number of values in the myarray array variable, and the addarray function adds
them.

Returning arrays from functions:

Passing an array variable from a function back to the shell script uses a similar technique.

The function uses an echo statement to output the individual array values in the proper
order, and the script must reassemble them into a new array variable:

```
$ cat test12
#!/bin/bash
# returning an array value
functionarraydblr {
localorigarray
localnewarray
local elements
locali
origarray=($(echo "$@"))
newarray=($(echo "$@"))
elements=$[ $# - 1 ]
for (( i = 0; i<= $elements; i++ ))
{
newarray[$i]=$[ ${origarray[$i]} * 2 ]
}

echo ${newarray[*]}
}
myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
arg1=$(echo ${myarray[*]})
result=($(arraydblr $arg1))
echo "The new array is: ${result[*]}"
$
$ ./test12
The original array is: 1 2 3 4 5
The new array is: 2 4 6 8 10
```

The script passes the array value, using the $arg1 variable to the arraydblr function.

The arraydblr function reassembles the array into a new array variable, and it makes a

copy for the output array variable. It then iterates through the individual array variable
values, doubles each value, and places it into the copy of the array variable in the function.
The arraydblr function then uses the echo statement to output the individual values

Function Recursion

One feature that local function variables provide is *self-containment.* A self-contained function
doesn't use any resources outside of the function, other than whatever variables the
script passes to it in the command line.
This feature enables the function to be called *recursively*, which means that the function calls itself to reach an answer. Usually, a recursive function has a base value that it eventually iterates down to. Many advanced mathematical algorithms use recursion to reduce a
complex equation down one level repeatedly, until they get to the level defined by the base
value.
The classic example of a recursive algorithm is calculating factorials. A factorial of a number is the value of the preceding numbers multiplied with the number. Thus, to find the
factorial of 5, you'd perform the following equation:
5! = 1 * 2 * 3 * 4 * 5 = 120
Using recursion, the equation is reduced down to the following format:
x! = x * (x-1)!
or in English, the factorial of *x* is equal to *x* times the factorial of *x-1*. This can be
expressed in a simple recursive script:

```
function factorial {
if [ $1 -eq 1 ]
then
echo 1
else
local temp=$[ $1 - 1 ]
local result='factorial $temp'
echo $[ $result * $1 ]
fi
}
```

The factorial function uses itself to calculate the value for the factorial:

```
$ cat test13
#!/bin/bash
```

```
# using recursion
function factorial {
if [ $1 -eq 1 ]
then
echo 1
else
local temp=$[ $1 - 1 ]
local result=$(factorial $temp)
echo $[ $result * $1 ]
fi
}
read -p "Enter value: " value
result=$(factorial $value)
echo "The factorial of $value is: $result"
$
$ ./test13
Enter value: 5
The factorial of 5 is: 120
$
```

Creating a Library:

It's easy to see how functions can help save typing in a single script, but what if you just happen to use the same single code block between scripts? It's obviously challenging if you have to defi ne the same function in each script, only to use it one time in each script.There's a solution for that problem! The bash shell allows you to create a library file for your functions and then reference that single library fi le in as many scripts as you need to.

```
$ catmyfuncs
# my script functions
functionaddem {
echo $[ $1 + $2 ]
}
functionmultem {
echo $[ $1 * $2 ]
}
functiondivem {
if [ $2 -ne 0 ]
then
echo $[ $1 / $2 ]
else
echo -1
fi
}
```

$
The next step is to include the myfuncs library fi le in your script fi les that want to use
any of the functions. This is where things get tricky.
The problem is with the scope of shell functions. As with environment variables, shell functions
are valid only for the shell session in which you defi ne them. If you run the myfuncs
shell script from your shell command line interface prompt, the shell creates a new shell
and runs the script in that new shell. This defines the three functions for that shell, but
when you try to run another script that uses those functions, they aren't available

Using Functions on the Command Line

You can use script functions to create some pretty complex operations. Sometimes, it would be nice to be able to use these functions directly on the command line interface prompt.Just as you can use a script function as a command in a shell script, you can also use a script function as a command in the command line interface. This is a nice feature because after you define the function in the shell, you can use it from any directory on the system;
you don't have to worry about a script being in your PATH environment variable. The trick
is to get the shell to recognize the function. You can do that in a couple of ways.

The fi rst method defines the function all on one line:

```
$ functiondivem { echo $[ $1 / $2 ]; }
$ divem 100 5
20
$
```

When you defi ne the function on the command line, you must remember to include a semicolon
at the end of each command, so the shell knows where to separate commands:

```
$ functiondoubleit { read -p "Enter value: " value; echo $[
$value * 2 ]; }
$
$ doubleit
Enter value: 20
40
$
```

The other method is to use multiple lines to defi ne the function. When you do that,
the bash shell uses the secondary prompt to prompt you for more commands. Using this
method, you don't need to place a semicolon at the end of each command; just press the
Enter key:

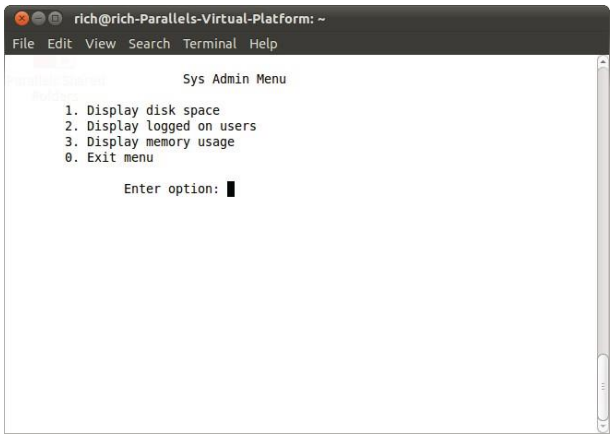
```
$ functionmultem {
>echo $[ $1 * $2 ]
> }
$ multem 2 5
10
$
```

When you use the brace at the end of the function, the shell knows that you're finished
defining the function.
Be extremely careful when creating functions on the command

writing script for graphical Desktop: Creating text menus-Building text windows:

Creating Text Menus

The most common way to create an interactive shell script is to utilize a menu. Offering your customers a choice of various options helps guide them through exactly what the script can and can't do.
Menu scripts usually clear the display area and then show a list of options available. The customer can select an option by pressing an associated letter or number assigned to each option. Figure 18-1
shows the layout of a sample menu.
The core of a shell script menu is the case command (see Chapter 12). The case command performs
specific commands, depending on what character your customer selects from the menu.
The following sections walk you through the steps you should follow to create a menu-based
shell script.



Create the menu layout

The first step in creating a menu is, obviously, to determine what elements you want to appear in the menu and lay them out the way that you want them to appear.

Before creating the menu, it's usually a good idea to clear the monitor display. This enables you to display your menu in a clean environment without distracting text. The clear command uses the terminfo data of your terminal session (see Chapter 2) to clear any text that appears on the monitor. After the clear command, you can use the echo command to display your menu elements.

By default, the echo command can only display printable text characters. When creating menu items, it's often helpful to use nonprintable items, such as the tab and newline characters.

To include these characters in your echo command, you must use the -e option.

Thus, the command:

```
echo -e "1.\tDisplay disk space"
```

results in the output line:

```
1. Display disk space
```

This greatly helps in formatting the layout of the menu items. With just a few echo commands, you can create a reasonable-looking menu:

```
clear
echo
echo -e "\t\t\tSys Admin Menu\n"
echo -e "\t1. Display disk space"
echo -e "\t2. Display logged on users"
```

```
echo -e "\t3. Display memory usage"
echo -e "\t0. Exit menu\n\n"
echo –en "\t\tEnter option: "
```

The -en option on the last line displays the line without adding the newline character atthe end. This gives the menu a more professional look, because the cursor stays at the end
of the line waiting for the customer's input.

The last part of creating the menu is to retrieve the input from the customer. This is done
using the read command (see Chapter 14). Because we expect only single-character input,
the nice thing to do is to use the -n option in the read command to retrieve only one character.

This allows the customer to enter a number without having to press the Enter key:

```
read -n 1 option
```

Next, you need to create your menu functions.

Create the menu functions

Shell script menu options are easier to create as a group of separate functions. This enables
you to create a simple, concise case command that is easy to follow.

To do that, you need to create separate shell functions for each of your menu options. The
first step in creating a menu shell script is to determine what functions you want your
script to perform and lay them out as separate functions in your code.

It is common practice to create stub functions for functions that aren't implemented yet. A
stub function is a function that doesn't contain any commands yet or possibly just an echo
statement indicating what should be there eventually:

```
functiondiskspace {
clear
echo "This is where the diskspace commands will go"
}
```

This enables your menu to operate smoothly while you work on the individual functions.

Create the menu functions

Shell script menu options are easier to create as a group of separate functions. This enables
you to create a simple, concise case command that is easy to follow.

To do that, you need to create separate shell functions for each of your menu options. The

first step in creating a menu shell script is to determine what functions you want your
script to perform and lay them out as separate functions in your code.
It is common practice to create stub functions for functions that aren't implemented yet. A
stub function is a function that doesn't contain any commands yet or possibly just an echo
statement indicating what should be there eventually:

```
functiondiskspace {
clear
echo "This is where the diskspace commands will go"
}
```

This enables your menu to operate smoothly while you work on the individual functions.
You don't have to code all the functions for your menu to work.The last part of creating the menu is to retrieve the input from the customer. This is done
using the read command (see Chapter 14). Because we expect only single-character input,
the nice thing to do is to use the -n option in the read command to retrieve only one character.
This allows the customer to enter a number without having to press the Enter key:

```
read -n 1 option
```

Next, you need to create your menu functions.
Create the menu functions
Shell script menu options are easier to create as a group of separate functions. This enables
you to create a simple, concise case command that is easy to follow.
To do that, you need to create separate shell functions for each of your menu options. The
first step in creating a menu shell script is to determine what functions you want your
script to perform and lay them out as separate functions in your code.
It is common practice to create stub functions for functions that aren't implemented yet. A
stub function is a function that doesn't contain any commands yet or possibly just an echo
statement indicating what should be there eventually:

```
functiondiskspace {
clear
echo "This is where the diskspace commands will go"
}
```

This enables your menu to operate smoothly while you work on the individual functions.
You don’t have to code all the functions for your menu to work.
One thing that helps out in the shell script menu is to create the menu layout itself as a
function:

```
function menu {
clear
echo
echo -e "\t\t\tSys Admin Menu\n"
echo -e "\t1. Display disk space"
echo -e "\t2. Display logged on users"
echo -e "\t3. Display memory usage"
echo -e "\t0. Exit program\n\n"
echo -en "\t\tEnter option: "
read -n 1 option
}
```

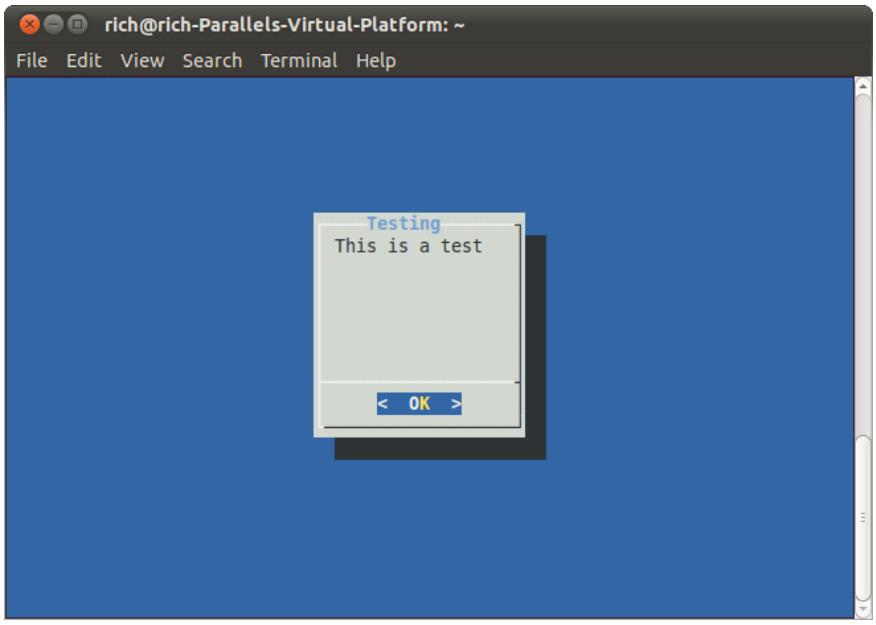
This enables you to easily redisplay the menu at any time just by calling the menu function.Add the menu logic
Now that you have your menu layout and your functions, you just need to create the
programming logic to put the two together. As mentioned, this requires the case command.
The case command should call the appropriate function according to the character selection
expected from the menu. It’s always a good idea to use the default case command
character (the asterisk) to catch any incorrect menu entries.
The following code illustrates the use of the case command in a typical menu:

```
menu
case $option in
0)
break ;;
1)
disk space ;;
2)
whose on ;;
3)
mem usage ;;
*)
clear
echo "Sorry, wrong selection";;
```

esac
This code first uses the menu function to clear the monitor screen and display the menu. The
read command in the menu function pauses until the customer hits a character on the keyboard.
Doing Windows
Using text menus is a step in the right direction, but there's still so much missing in
our interactive scripts, especially if we try to compare them to the graphical Windows
world. Fortunately for us, some very resourceful people out in the open source world have
helped us out.
The dialog package is a nifty little tool originally created by Savio Lam and currently maintained
by Thomas E. Dickey. This package recreates standard Windows dialog boxes in a
text environment using ANSI escape control codes. You can easily incorporate these dialog
boxes in your shell scripts to interact with your script users. This section describes the dialog
package and demonstrates how to use it in shell scripts.
The dialog package
The dialog command uses command line parameters to determine what type of Windows
*widget*to produce. A widget is the dialog package term for a type of Windows element. The
dialog package currently supports the types of widgets

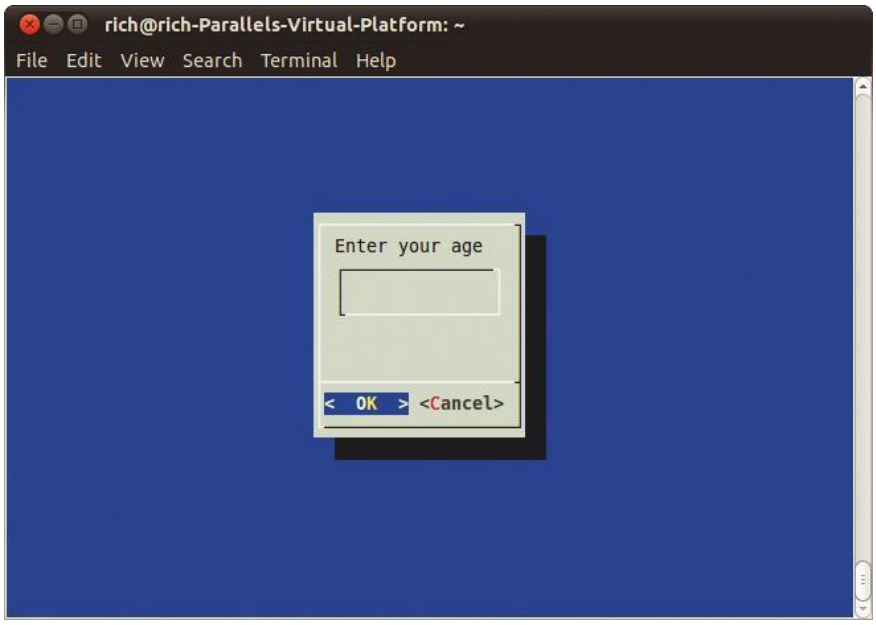
The dialog Widgets
Widget Description
calendar-Provides a calendar from which to select a date
checklist- Displays multiple entries where each entry can be turned on or off
form-Allows you to build a form with labels and text fields to be fi lled out
fselect-Provides a file selection window to browse for a fi le
gauge-Displays a meter showing a percentage of completion
infobox-Displays a message without waiting for a response
inputbox-Displays a single text form box for text entry
inputmenu- Provides an editable menu
menu-Displays a list of selections from which to choose

As you can see from Table 18-1, you can choose from lots of different widgets. This can give
your scripts a more professional look with very little effort.
To specify a specifi c widget on the command line, you need to use the double dash format:
dialog --widget parameters
where widget is the widget name as seen in Table 18-1, and parameters defines the size
of the widget window and any text required for the widget.
Each dialog widget provides output in two forms:
■ Using STDERR
■ Using the exit code status
The exit code status of the dialog command determines the button selected by the
user. If an OK or Yes button is selected, the dialog command returns a 0 exit status.
If a Cancel or No button is selected, the dialog command returns a 1 exit status. You
can use the standard $? variable to determine which button was selected in the dialog
widget.
If a widget returns any data, such as a menu selection, the dialog command sends the
data to STDERR. You can use the standard bash shell technique of redirecting the STDERR
output to another fi le or fi le descriptor:
dialog --inputbox "Enter your age:" 10 20 2>age.txt
This command redirects the text entered in the textbox to the age.txt fi le.
The following sections look at some examples of the more common dialog widgets you'll use



If your terminal emulator supports the mouse, you can click the OK button to close the dialog box. You can also use keyboard commands to simulate a click — just press the Enter key.

The exit status of the dialog command is set depending on which button the user selects.
If the No button is selected, the exit status is 1, and if the Yes button is selected, the exit
status is 0.
The inputbox widget:
The inputbox widget provides a simple textbox area for the user to enter a text string.
The dialog command sends the value of the text string to STDERR. You must redirect that
to retrieve the answer. Figure 18-4 demonstrates what the inputbox widget looks like.



As you can see in Figure 18-4, the inputboxprovides two buttons — OK and Cancel. If the Cancel button is selected, the exit status of the command is 1; otherwise, the exit status is 0:

```
$ dialog --inputbox "Enter your age:" 10 20 2>age.txt
$ echo $?
0
$ cat age.txt
12$
```

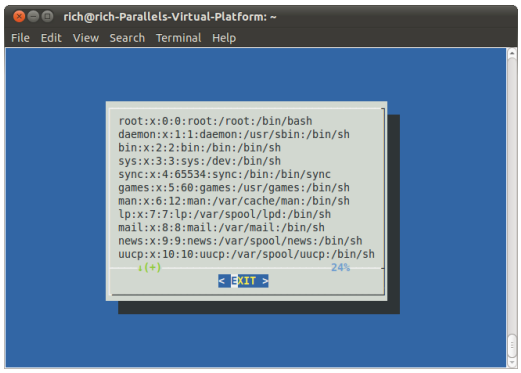
You'll notice when you use the cat command to display the contents of the text fi le that there's no newline character after the value. This enables you to easily redirect the fi le contents to a variable in a shell script to extract the string entered by the user.

The textbox widget

The textbox widget is a great way to display lots of information in a window. It produces a scrollable window containing the text from a fi le specified in the parameters:

```
$ dialog --textbox /etc/passwd 15 45
```

The contents of the /etc/passwd fi le are shown within the scrollable text window, as illustrated in Figure 18-5.



You can use the arrow keys to scroll left and right, as well as up and down in the text
fi le. The bottom line in the window shows the percent location within the fi le that
you’re viewing. The textbox contains only a single Exit button, which should be selected
to exit the widget.

The menu widget

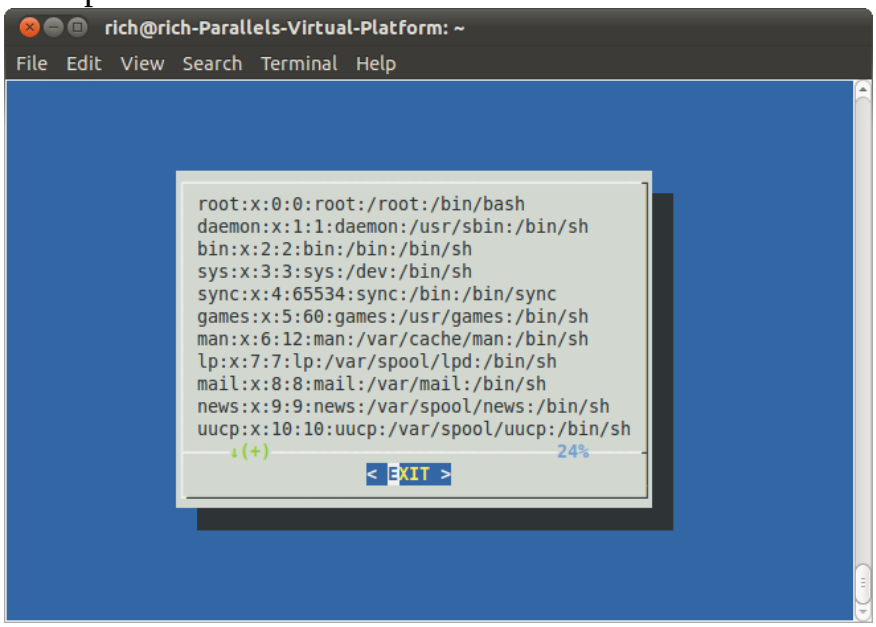
The menu widget allows you to create a window version of the text menu we created earlier
in this chapter. You simply provide a selection tag and the text for each item:

```
$ dialog --menu "Sys Admin Menu" 20 30 10 1 "Display disk space"
2 "Display users" 3 "Display memory usage" 4 "Exit" 2> test.txt
```

The fi rst parameter defines a title for the menu. The next two parameters defi ne the height
and width of the menu window, while the third parameter defines the number of menu
items that appear in the window at one time. If there are more menu items, you can scroll
through them using the arrow keys.

Following those parameters, you must add menu item pairs. The fi rst element is the tag
used to select the menu item. Each tag should be unique for each menu item and can be
selected by pressing the appropriate key on the keyboard. The second element is the text

used in the menu. Figure 18-6 demonstrates the menu produced by the example command



The menu widget

The menu widget allows you to create a window version of the text menu we created earlier

in this chapter. You simply provide a selection tag and the text for each item:

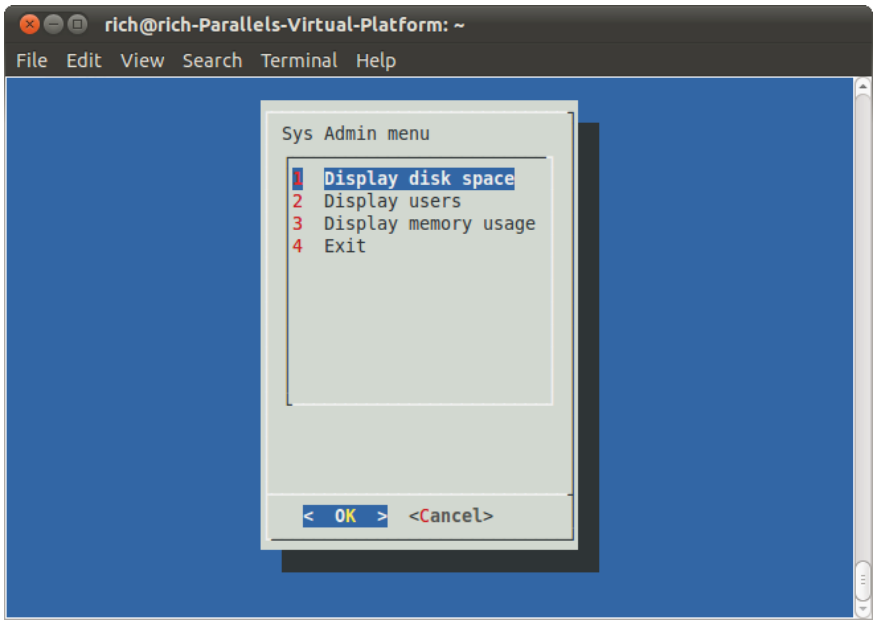
$ dialog --menu "Sys Admin Menu" 20 30 10 1 "Display disk space"
2 "Display users" 3 "Display memory usage" 4 "Exit" 2> test.txt

The fi rst parameter defines a title for the menu. The next two parameters defi ne the height

and width of the menu window, while the third parameter defines the number of menu

items that appear in the window at one time. If there are more menu items, you can scroll

through them using the arrow keys.



The fselect widget

There are several fancy built-in widgets provided by the dialog command. The fselect

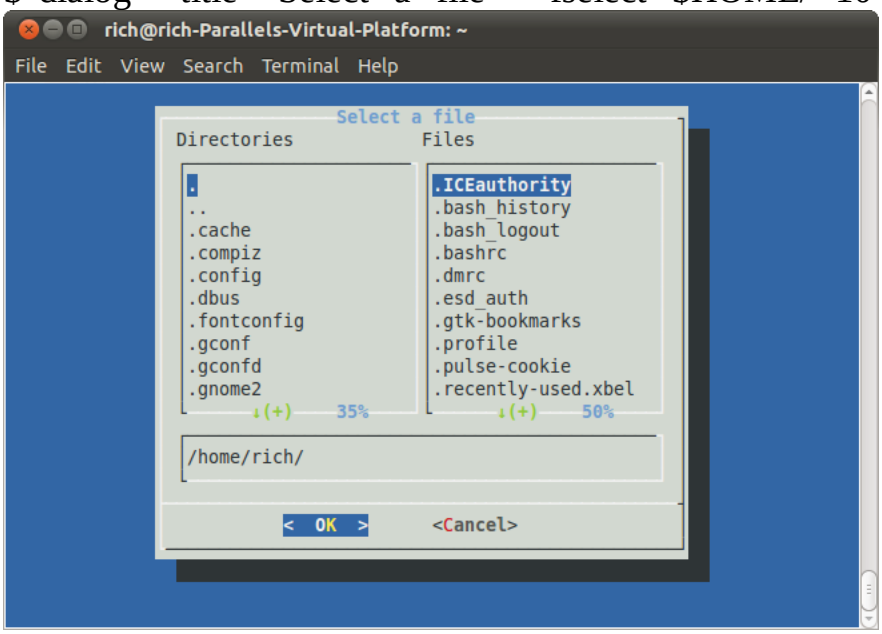
widget is extremely handy when working with fi lenames. Instead of forcing the user to

type a fi lename, you can use the fselect widget to browse to the fi le location and select

the fi le, as shown in Figure 18-7.

The fselect widget format looks like:

$ dialog --title "Select a file" --fselect $HOME/ 10 50 2>file.txt



The fi rst parameter after the fselect option is the starting folder location used in the window. The fselect widget window consists of a directory listing on the left side, a fi le

listing on the right side that shows all the fi les in the selected directory, and a simple textbox
that contains the currently selected fi le or directory. You canmanualtype a fi lename
in the textbox, or you can use the directory and fi le listings to select one (use the spacebar
to select a fi le to add to the textbox).

The dialog options
In addition to the standard widgets, you can customize lots of different options in the dialog
command. You've already seen the --title parameter in action. This allows you to set
a title for the widget that appears at the top of the window.
Lots of other options allow you to completely customize both the appearance and the
behavior of your windows. Table 18-2 shows the options available for the dialog
command.
TABLE 18-2 The dialog Command Options
Option Description
--add-widget Proceeds to the next dialog unless Esc or the Cancel button has been pressed
--aspect ratio Specifies the width/height aspect ratio of the window
--backtitle title Specifies a title to display on the background, at the top of the
screen
--begin x y Specifies the starting location of the top-left corner of the window
--cancel-label label Specifies an alternative label for the Cancel button
Continues
Option Description
--clear Clears the display using the default dialog background color
--colors Embeds ANSI color codes in dialog text
--cr-wrap Allows newline characters in dialog text and forces a line wrap
--create-rc file Dumps a sample configuration fi le to the specified fi le
--defaultno Makes the default of a yes/no dialog No
--default-item string Sets the default item in a checklist, form, or menu dialog
--exit-label label Specifies an alternative label for the Exit button

--extra-button Displays an extra button between the OK and Cancel buttons
--extra-label label Specifies an alternative label for the Extra button
--help Displays the dialog command help message
--help-button Displays a Help button after the OK and Cancel buttons
--help-label label Specifies an alternative label for the Help button
--help-status Writes the checklist, radiolist, or form information after the help information in the Help button was selected
--ignore Ignores options that dialog does not recognize
--input-fdfdSpecifies an alternative fi le descriptor, other than STDIN
--insecure Changes the password widget to display asterisks when typing
--item-help Adds a help column at the bottom of the screen for each tag in a checklist, radiolist, or menu for the tag item
--keep-window Doesn't clear old widgets from the screen
--max-input size Specifies a maximum string size for the input; default is 2048
--nocancelSuppresses the Cancel button
--no-collapse Doesn't convert tabs to spaces in dialog text
--no-kill Places the tailboxbg dialog in background and disables SIGHUP for the process
--no-label label Specifies an alternative label for the No button
--no-shadow Doesn't display shadows for dialog windows
--ok-label label Specifies an alternative label for the OK button
--output-fdfdSpecifies an alternative output fi le descriptor other than STDERR
--print-maxsize Prints the maximum size of dialog windows allowed to the
output

Introducing sed and gaw:

Getting to know the sed editor

The sed editor is called a *stream editor*, as opposed to a normal interactive text editor. In an interactive
text editor, such as vim, you interactively use keyboard commands to insert, delete, or replace text in the data. A stream editor edits a stream of data based on a set of rules you supply ahead of time, before the editor processes the data

After the stream editor matches all the commands against a line of data, it reads the next
line of data and repeats the process. After the stream editor processes all the lines of data
in the stream, it terminates.

Because the commands are applied sequentially line by line, the sed editor makes only one
pass through the data stream to make the edits. This makes the sed editor much faster than
an interactive editor and allows you to quickly make changes to data in a fi le on the fl y.
Here's the format for using the sed command:
sed options script file
The options parameters allow you to customize the behavior of the sed command and
include the options shown in Table 19-1.
TABLE 19-1 Thesed Command Options
Option Description
-e script Adds commands specified in the script to the commands run while processing
the input
-f file Adds the commands specified in the fi le to the commands run while processing
the input
-n Doesn't produce output for each command, but waits for the print command
The script parameter specifies a single command to apply against the stream data. If more
than one command is required, you must use either the -e option to specify them in the
command line or the -f option to specify them in a separate fi le. Numerous commands are
available for manipulating data. We examine some of the basic commands used by the sed
editor in this chapter and then look at some of the more advanced commands
Defining an editor command in the command line
By default, the sed editor applies the specified commands to the STDIN input stream. This
allows you to pipe data directly to the sed editor for processing. Here's a quick example
demonstrating how to do this:When you run this example, it should display the results almost instantaneously. That's the
power of using the sed editor. You can make multiple edits to data in about the same time
it takes for some of the interactive editors just to start up.
Of course, this simple test just edited one line of data. You should get the same speedy

results when editing complete fi les of data:

```
$ cat data1.txt
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
$
$ sed 's/dog/cat/' data1.txt
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
$
```

The sed command executes and returns the data almost instantaneously. As it processes each line of data, the results are displayed. You'll start seeing results before the sed editor completes processing the entire fi le.

It's important to note that the sed editor doesn't modify the data in the text fi le itself. It only sends the modified text to STDOUT. If you look at the text fi le, it still contains the original data:

```
$ cat data1.txt
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
$
```

Using multiple editor commands in the command line

To execute more than one command from the sed command line, just use the -e option:

```
$ sed -e 's/brown/green/; s/dog/cat/' data1.txt
The quick green fox jumps over the lazy cat.
```

Instead of using a semicolon to separate the commands, you can use the secondary prompt in the bash shell. Just enter the fi rst single quotation mark to open the sedprogram script (sededitor command list), and bash continues to prompt you for more commands until you enter the closing quotation mark:

```
$ sed -e '
>s/brown/green/
>s/fox/elephant/
>s/dog/cat/' data1.txt
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
$
```

You must remember to fi nish the command on the same line where the closing single quotation

mark appears. After the bash shell detects the closing quotation mark, it processes the command. After it starts, the sedcommand applies each command you specified to each line of data in the text fi le.

Reading editor commands from a file

Finally, if you have lots of sedcommands you want to process, it is often easier to just store them in a separate fi le. Use the -f option to specify the fi le in the sedcommand:

```
$ cat script1.sed
s/brown/green/
s/fox/elephant/
s/dog/cat/
$
$ sed -f script1.sed data1.txt
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
$
```

In this case, you don't put a semicolon after each command. The sededitor knows that each line contains a separate command. As with entering commands on the command line,Instead of using a semicolon to separate the commands, you can use the secondary prompt in the bash shell. Just enter the fi rst single quotation mark to open the sed program script (sed editor command list), and bash continues to prompt you for more commands until you enter the closing quotation mark:

```
$ sed -e '
>s/brown/green/
>s/fox/elephant/
> s/dog/cat/' data1.txt
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
$
```

You must remember to fi nish the command on the same line where the closing single quotation mark appears. After the bash shell detects the closing quotation mark, it processes the command. After it starts, the sed command applies each command you specified to each line of data in the text fi le.

Reading editor commands from a file

Finally, if you have lots of sed commands you want to process, it is often easier to just store them in a separate fi le. Use the -f option to specify the fi le in the sed command:

```
$ cat script1.sed
s/brown/green/
s/fox/elephant/
s/dog/cat/
$
$ sed -f script1.sed data1.txt
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
$
```

In this case, you don't put a semicolon after each command. The sed editor knows that each line contains a separate command. As with entering commands on the command line,

Commanding at the sed Editor Basics

The key to successfully using the sed editor is to know its myriad of commands and formats,

which help you to customize your text editing. This section describes some of the

basic commands and features you can incorporate into your script to start using the sed

editor.

Introducing more substitution options

You've already seen how to use the s command to substitute new text for the text in a line.

However, a few additional options are available for the substitute command that can help make your life easier.

Substituting flags

There's a caveat to how the substitute command replaces matching patterns in the text string. Watch what happens in this example:

```
$ cat data4.txt
This is a test of the test script.
This is the second test of the test script.
$
$ sed 's/test/trial/' data4.txt
This is a trial of the test script.
This is the second trial of the test script.
$
```

- A number, indicating the pattern occurrence for which new text should be substituted
- g, indicating that new text should be substituted for all occurrences of the existing text
- p, indicating that the contents of the original line should be printed
- w *file*, which means to write the results of the substitution to a file

In the first type of substitution, you can specify which occurrence of the matching pattern the sed editor should substitute new text for:

```
$ sed 's/test/trial/2' data4.txt
This is a test of the trial script.
This is the second test of the trial script.
$
```

As a result of specifying a 2 as the substitution flag, the sed editor replaces the pattern only in the second occurrence in each line. The g substitution flag enables you to replace every occurrence of the pattern in the text:

```
$ sed 's/test/trial/g' data4.txt
This is a trial of the trial script.
This is the second trial of the trial script.
$
```

The p substitution flag prints a line that contains a matching pattern in the substitute command. This is most often used in conjunction with the -n sed option:

```
$ cat data5.txt
This is a  test  line.
This is a different line.
$
$ sed -n 's/test/trial/p' data5.txt
This is a trial line.
$
```

The -n option suppresses output from the sed editor. However, the p substitution flag outputs any line that has been modified. Using the two in combination produces output only for lines that have been modified by the substitute command.

Replacing characters

Sometimes, you run across characters in text strings that aren't easy to use in the substitution pattern. One popular example in the Linux world is the forward slash (/).

Substituting pathnames in a fi le can get awkward. For example, if you wanted to substitute
the C shell for the bash shell in the /etc/passwd fi le, you'd have to do this:
$ sed 's/\/bin\/bash/\/bin\/csh/' /etc/passwd
Because the forward slash is used as the string delimiter, you must use a backslash to
escape it if it appears in the pattern text. This often leads to confusion and mistakes.
To solve this problem, the sed editor allows you to select a different character for the
string delimiter in the substitute command:
$ sed 's!/bin/bash!/bin/csh!' /etc/passwd
In this example, the exclamation point is used for the string delimiter, making the pathnames
much easier to read and understand.
Using addresses
By default, the commands you use in the sed editor apply to all lines of the text data. If
you want to apply a command only to a specifi c line or a group of lines, you must use line
addressing.
There are two forms of line addressing in the sed editor:
■ A numeric range of lines
■ A text pattern that fi lters out a line
Both forms use the same format for specifying the address:
[address]command

You can also group more than one command together for a specifi c address:
address {
command1
command2
command3
}
The sededitor applies each of the commands you specify only to lines that match the addressspecifi ed. This section demonstrates using both of these addressing techniques in yoursededitor scripts.

Addressing the numeric line

When using numeric line addressing, you reference lines using their line position in the text stream. The sededitor assigns the fi rst line in the text stream as line number one and continues sequentially for each new line.
The address you specify in the command can be a single line number or a range of lines specified by a starting line number, a comma, and an ending line number. Here's an example of specifying a line number to which the sedcommand will be applied:

```
$ sed '2s/dog/cat/' data1.txt
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
$
```

The sededitor modified the text only in line two per the address specifi ed. Here's another

example, this time using a range of line addresses:

```
$ sed '2,3s/dog/cat/' data1.txt
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy dog
$
```

If you want to apply a command to a group of lines starting at some point within the text, but continuing to the end of the text, you can use the special address, the dollar sign:

```
$ sed '2,$s/dog/cat/' data1.txt
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy cat
```

Regular Expressions:

Linux Regular Expressions are special characters which help search data and matching complex patterns. Regular expressions are shortened as ‘regexp’ or ‘regex’. They are used in many Linux programs like grep, bash, rename, sed, etc.

Types of Regular expressions

For ease of understanding let us learn the different types of Regex one by one.

Basic Regular expressions

Some of the commonly used commands with Regular expressions are tr, sed, vi and grep. Listed below are some of the basic Regex.

Symbol	Descriptions
.	replaces any character
^	matches start of string
$	matches end of string
*	matches up zero or more times the preceding character
\	Represent special characters
()	Groups regular expressions
?	Matches up exactly one character

Regular expressions in Linux, commonly referred to as Reg Ex, are powerful tools used for pattern matching and text manipulation in Linux and other programming languages. Regular expressions allow users to define patterns that can match specific strings of characters or patterns within larger strings of text.

Defining regular Expressions:

The Linux regular expression, basically it is a sequence of characters or string that would define the searching pattern. These searching patterns are used by the string search algorithms like vim, vi, sed, awk, find, grep, etc. It is a very powerful tool in Linux.

Interval Regular expressions

These expressions tell us about the number of occurrences of a character in a string. They are

Expression	Description
{n}	Matches the preceding character appearing ‘n’ times exactly

Expression	Description
{n,m}	Matches the preceding character appearing 'n' times but not more than m
{n, }	Matches the preceding character only when it appears 'n' times or more

Example:

Filter out all lines that contain character 'p'

```
guru99@guru99-VirtualBox:~$ cat sample|grep p
apple
pant
people
```

We want to check that the character 'p' appears exactly 2 times in a string one after the other. For this the syntax would be:

cat sample | grep -E p\{2}

```
guru99@guru99-VirtualBox:~$ cat sample|grep -E p\{2}
apple
guru99@guru99-VirtualBox:~$
```

Note: You need to add -E with these regular expressions.

Extended regular expressions

These regular expressions contain combinations of more than one expression. Some of them are:

Expression	Description
\+	Matches one or more occurrence of the previous character
\?	Matches zero or one occurrence of the previous character

Example:

Searching for all characters 't'

```
guru99@guru99-VirtualBox:~$ cat sample|grep t
bat
ant
eat
pant
taste
```

Suppose we want to filter out lines where character 'a' precedes character 't'

We can use command like

catsample|grep "a\+t"

```
guru99@guru99-VirtualBox:~$ cat sample|grep "a\+t"
bat
eat
guru99@guru99-VirtualBox:~$
```

Brace expansion

The syntax for brace expansion is either a sequence or a comma separated list of items inside curly braces "{}". The starting and ending items in a sequence are separated by two periods "..".

Interval Regular expressions

These expressions tell us about the number of occurrences of a character in a string. They are

Expression	Description
{n}	Matches the preceding character appearing 'n' times exactly
{n,m}	Matches the preceding character appearing 'n' times but not more than m
{n, }	Matches the preceding character only when it appears 'n' times or more

Example:

Filter out all lines that contain character 'p'

```
guru99@guru99-VirtualBox:~$ cat sample|grep p
apple
pant
people
```

We want to check that the character 'p' appears exactly 2 times in a string one after the other. For this the syntax would be:

cat sample | grep -E p\{2}

```
guru99@guru99-VirtualBox:~$ cat sample|grep -E p\{2}
apple
guru99@guru99-VirtualBox:~$
```

Note: You need to add -E with these regular expressions.

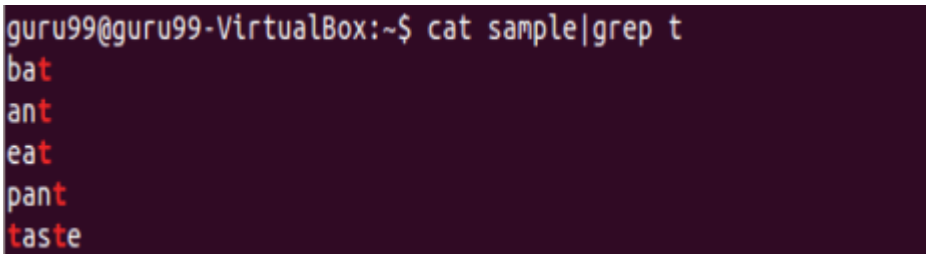
Extended regular expressions

These regular expressions contain combinations of more than one expression. Some of them are:

Expression	Description
\+	Matches one or more occurrence of the previous character
\?	Matches zero or one occurrence of the previous character

Example:

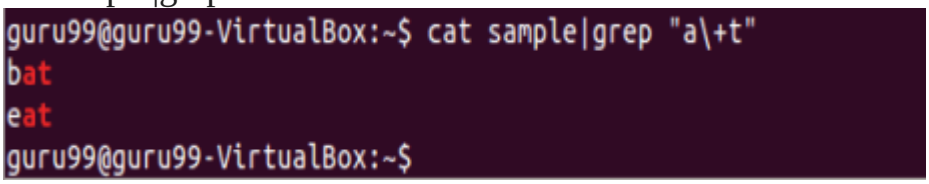
Searching for all characters ‘t’



Suppose we want to filter out lines where character ‘a’ precedes character ‘t’

We can use command like

catsample|grep "a\+t"



Brace expansion

The syntax for brace expansion is either a sequence or a comma separated list of items inside curly braces “{}”. The starting and ending items in a sequence are separated by two periods “..”.