Unit III

Creating Functions

- *Functions* are blocks of script code that 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)

- There are two formats you can use to create functions in bash shell scripts. The first format uses the keyword `function`, along with the function name you assign to the block of code:

Creating a function

- ```
  function name {
  commands
  }
  ```

The *name* attribute defines a unique name assigned to the function. Each function you define in your script must be assigned a unique name.
  ```

- The second format for defining a function in a bash shell script more closely follows how functions are defined in other programming languages:

- ```
  name() {
  commands

  }
  ```

Using functions :

```
$ 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
$
```

- You also need to be careful about your function names. Remember, each function name must be unique, or you'll have a problem. If you redefine a function, the new definition overrides the original function definition, without producing any error messages:

```
$ cat test3
#!/bin/bash
# testing using a duplicate function name

function func1 {
echo "This is the first definition of the function name"
}

func1

function func1 {
   echo "This is a repeat of the same function name"
}

func1
echo "This is the end of the script"
$
$ ./test3
This is the first definition of the function name
This is a repeat of the same function name
This is the end of the script
$
```

The original definition of the func1 function works fine, but after the second definition of the func1 function, any subsequent uses of the function use the second definition.

- **Returning a Value :**

```
$ cat test5b
#!/bin/bash
# using the echo to return a value

function dbl {
   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
```

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

- **Using Variables in Functions:**
  Passing parameters to a function:
  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

function addem {
    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 first checks the number of parameters passed to it 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 to itself for the result. If there are two parameters, addem adds them together for the result.

```
$ 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
$
```

Handling variables in a function:

- One thing that causes problems for shell script programmers is the *scope of a variable.*
- *The* scope is where the variable is visible. Variables defined in functions can have a different scope than regular variables. That is, they can be hidden from the rest of the script.
- Functions use two types of variables:
  - Global
  - Local

- **Global variables**：
  Global variables are variables that are valid anywhere within the shell script. If you define a global variable in the main section of a script, you can retrieve its value inside a function. Likewise, if you define a global variable inside a function, you can retrieve its value in the main section of the script.
- By default, any variables you define in the script are global variables. Variables defined outside of a function can be accessed within the function just fine:

```
$ cat test8
#!/bin/bash
# using a global variable to pass a value

function dbl {
   value=$[ $value * 2 ]
}

read -p "Enter a value: " value
dbl
echo "The new value is: $value"
$
$ ./test8
Enter a value: 450
The new value is: 900
$
```

The `$value` variable is defined outside of the function and assigned a value outside of the function. When the `dbl` function is called, the variable and its value are still valid inside the function. When the variable is assigned a new value inside the function, that new value is still valid when the script references the variable.

- This can be a dangerous practice, however, especially if you intend to use your functions in different shell scripts. It requires that you know exactly what variables are used in the function, including any variables used to calculate values not returned to the script. Here's an example of how things can go bad:

```
$ cat badtest2
#!/bin/bash
# demonstrating a bad use of variables

function func1 {
    temp=$[ $value + 5 ]
    result=$[ $temp * 2 ]
}

temp=4
value=6

func1
echo "The result is $result"
if [ $temp -gt $value ]
then
    echo "temp is larger"
else
    echo "temp is smaller"
fi
$
$ ./badtest2
The result is 22
temp is larger
$
```

- **Local variables**

  use the `local` keyword in front of the variable declaration: `local temp`

```
$ cat test9
#!/bin/bash
# demonstrating the local keyword

function func1 {
   local temp=$[ $value + 5 ]
   result=$[ $temp * 2 ]
}

temp=4
value=6

func1
echo "The result is $result"
if [ $temp -gt $value ]
then
   echo "temp is larger"
else
   echo "temp is smaller"
fi
$
$ ./test9
The result is 22
temp is smaller
$
```

- **Array Variables and Functions:**
- **Passing arrays to functions:**
  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.

```
# adding values in an array

function addarray {
   local sum=0
   local newarray
   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
$
```

- **Returning arrays from functions:**

```
$ cat test12
#!/bin/bash
# returning an array value

function arraydblr {
   local origarray
   local newarray
   local elements
   local i
   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
```

- **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 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:

The first step in the process is to create a common library file that contains the functions you need in your scripts. Here's a simple library file called `myfuncs` that defines three simple functions:

```
$ cat myfuncs
# my script functions

function addem {
   echo $[ $1 + $2 ]
}

function multem {
   echo $[ $1 * $2 ]
}

function divem {
   if [ $2 -ne 0 ]
   then
      echo $[ $1 / $2 ]
   else
      echo -1
   fi
}
$
```

- This example assumes that the `myfuncs` library file is located in the same directory as the shell script. If not, you need to use the appropriate path to access the file. Here's an example of creating a script that uses the myfuncs library file:

```
$ cat test14
#!/bin/bash
# using functions defined in a library file
. ./myfuncs

value1=10
value2=5
result1=$(addem $value1 $value2)
result2=$(multem $value1 $value2)
result3=$(divem $value1 $value2)
echo "The result of adding them is: $result1"
echo "The result of multiplying them is: $result2"
echo "The result of dividing them is: $result3"
$
$ ./test14
The result of adding them is: 15
The result of multiplying them is: 50
The result of dividing them is: 2
$
```

The script successfully uses the functions defined in the `myfuncs` library file.

- **Using Functions on the Command Line:**
  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;

  Creating functions on the command line : Because the shell interprets commands as you type them, you can define a function directly on the command line. You can do that in two ways.

The first method defines the function all on one line:

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

When you define the function on the command line, you must remember to include a semi-colon at the end of each command, so the shell knows where to separate commands:

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

- The other method is to use multiple lines to define 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:

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

Writing Scripts for Graphical Desktops

- **Creating Text Menus:**

FIGURE 18-1

Displaying a menu from a 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 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"`

```
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 at the 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.

- **Create the menu functions:**
  Add the menu logic:

```
$ cat menu1
#!/bin/bash
# simple script menu

function diskspace {
   clear
   df -k
}

function whoseon {
   clear
   who
}

function memusage {
   clear
   cat /proc/meminfo
}

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
}
```

```
while [ 1 ]
do
   menu
   case $option in
   0)
      break ;;
   1)
      diskspace ;;


   2)
      whoseon ;;
   3)
      memusage ;;
   *)
      clear
      echo "Sorry, wrong selection";;
   esac
   echo -en "\n\n\t\t\tHit any key to continue"
   read -n 1 line
done
clear
$
```

- Doing Windows: The *dialog* package is a nifty little tool originally **created by Savio Lam** and currently **maintained by Thomas E. Dickey.**
  The dialog package:

- To specify a specific widget on the command line, you need to use the double dash format:

  `dialog --`*`widget parameters`*

  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 file or file descriptor:

  ```
  dialog --inputbox "Enter your age:" 10 20 2>age.txt
  ```

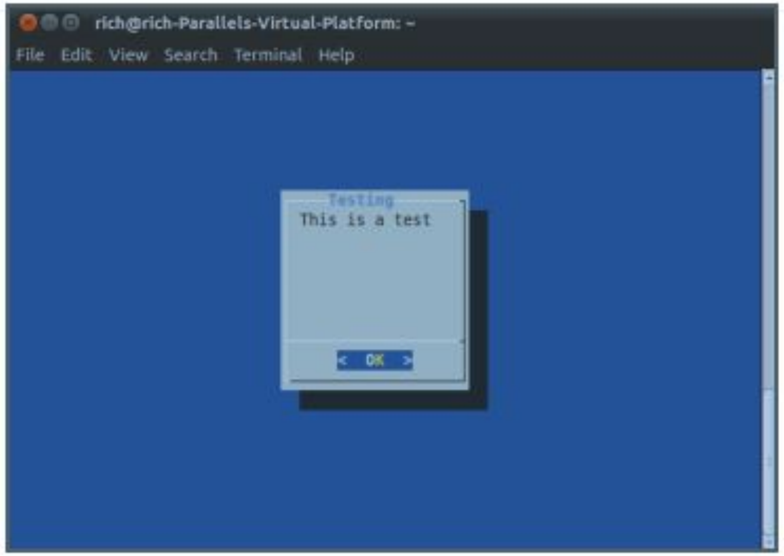
  This command redirects the text entered in the textbox to the `age.txt` file.

- **The msgbox widget:**
  The `msgbox` widget is the most common type of dialog box.
  `dialog --msgbox` *`text height width`*

- The *`text`* parameter is any string you want to place in the window. The `dialog` command automatically wraps the text to fit the size of the window you create, using the *`height`* and *`width`* parameters.

FIGURE 18-2

Using the msgbox widget in the dialog command



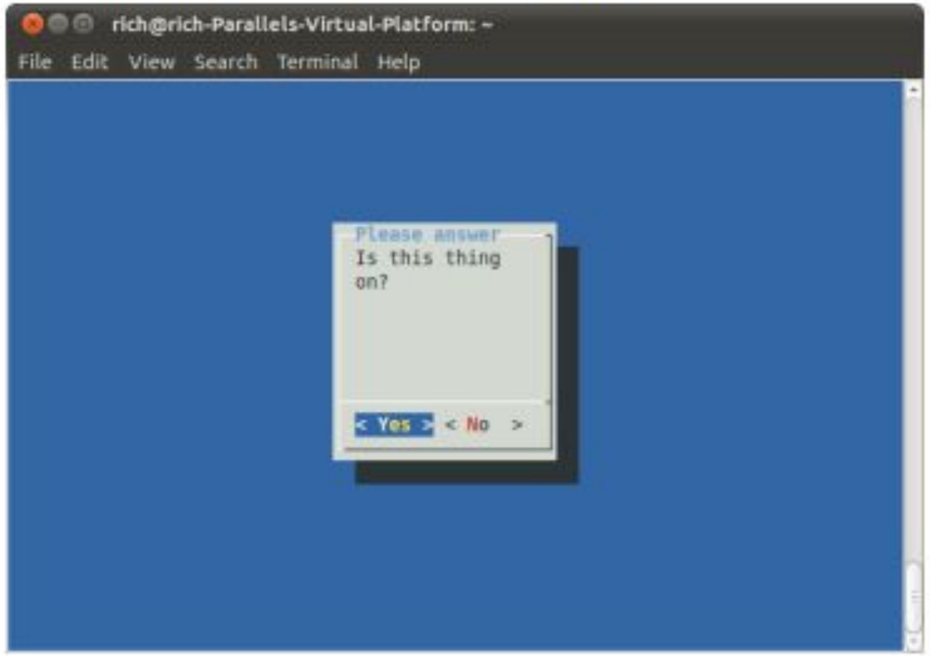
- If your terminal emulator supports the mouse, you can click the OK button to close the dialog box. You can also use keyboard commands to simulate a click — just press the Enter key.

- **The yesno widget:**
  Here's an example of using the `yesno` widget:

```
$ dialog --title "Please answer" --yesno "Is this thing on?" 10 20
$ echo $?
1

$
```

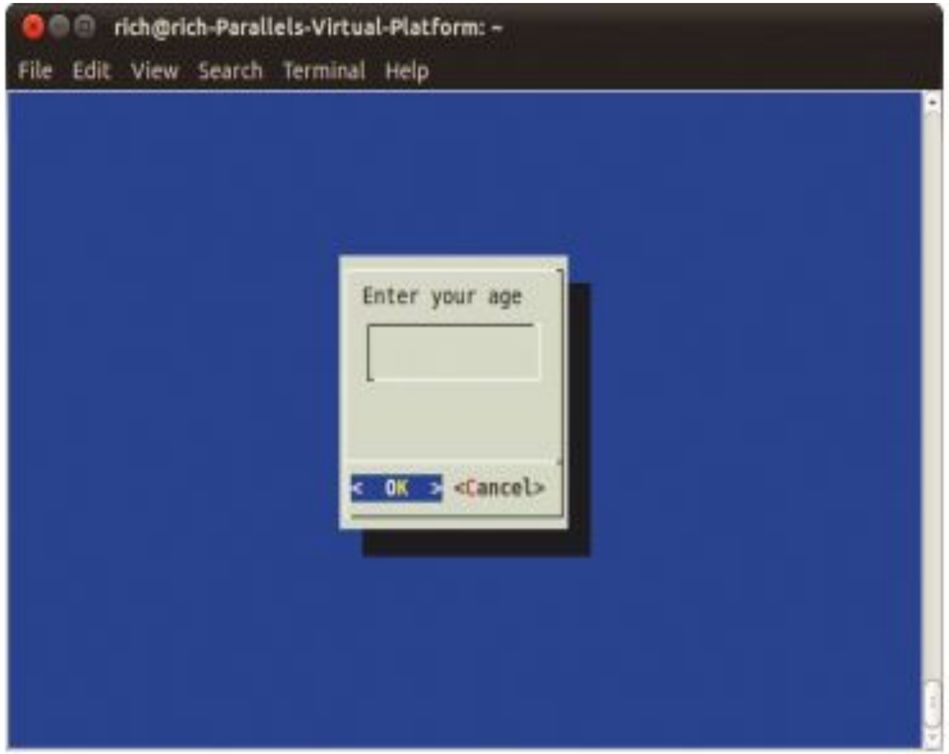
FIGURE 18-3

Using the yesno widget in the dialog command



- **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.

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

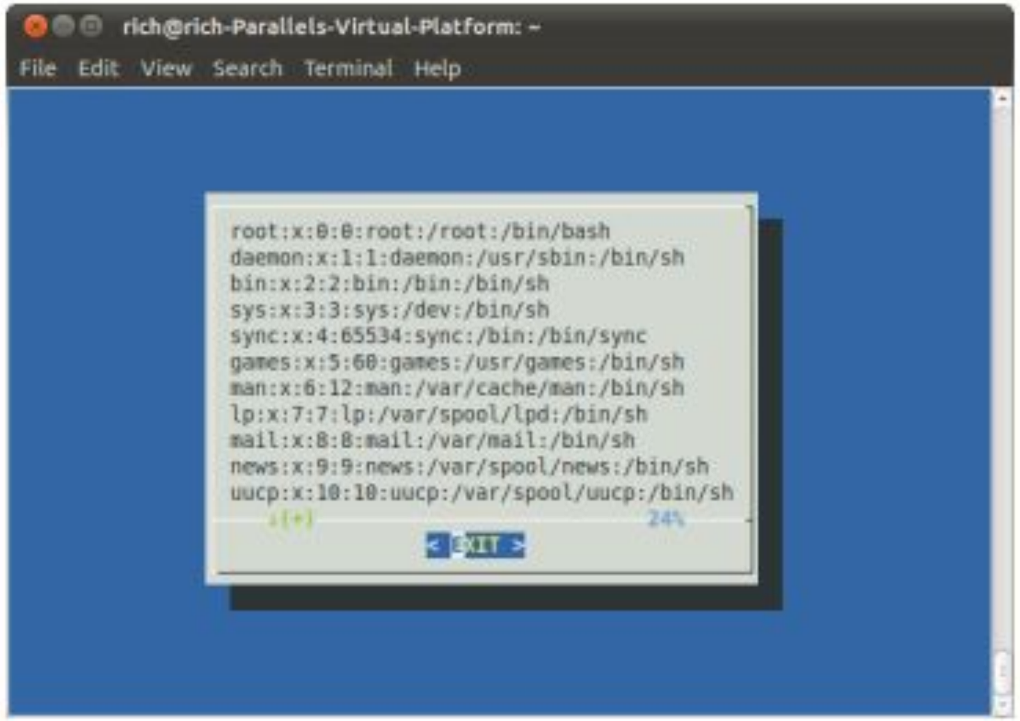


- The `inputbox` provides two buttons — OK and Cancel. If the Cancel button is selected, the exit status of the command is 1; otherwise, the exit status is 0:

- **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 file specified in the parameters:
  `$ dialog --textbox /etc/passwd 15 45`

  The contents of the `/etc/passwd` file are shown within the scrollable text window

The textbox widget



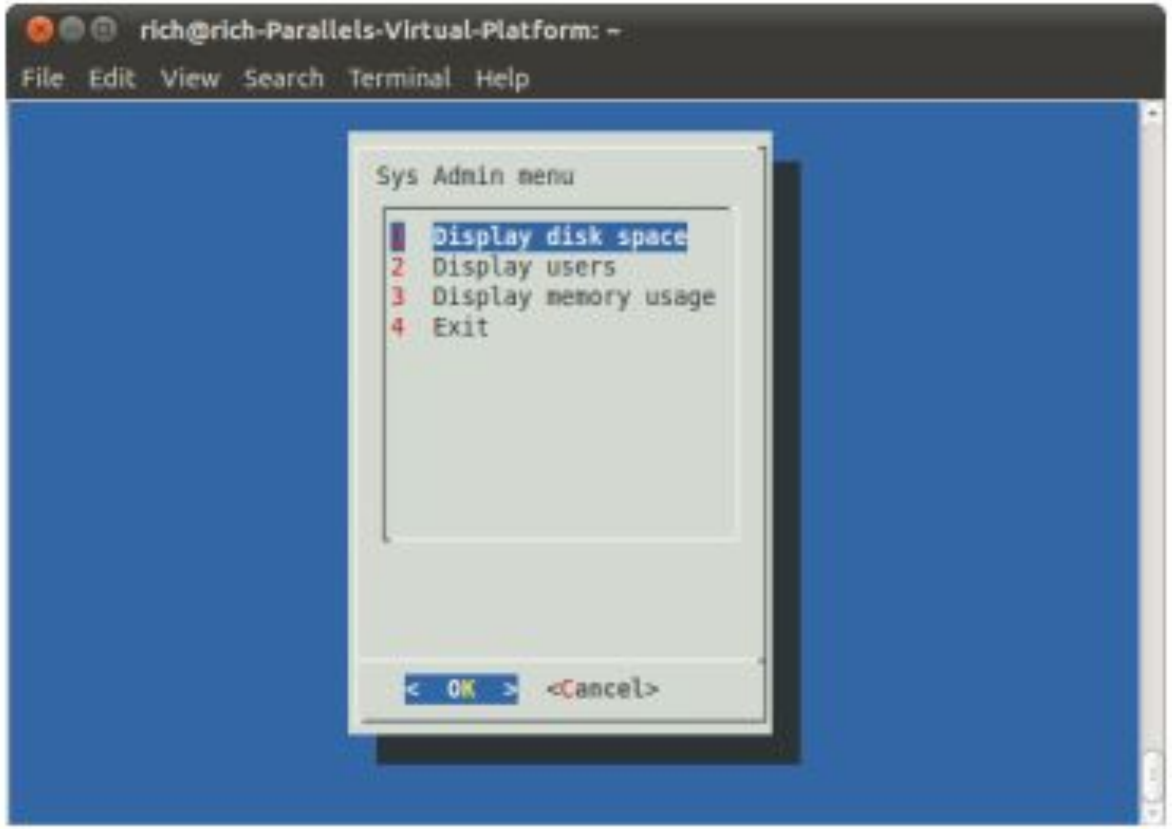
- **The menu widget:**

  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 first parameter defines a title for the menu.
- The next two parameters define 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 menu widget with menu items



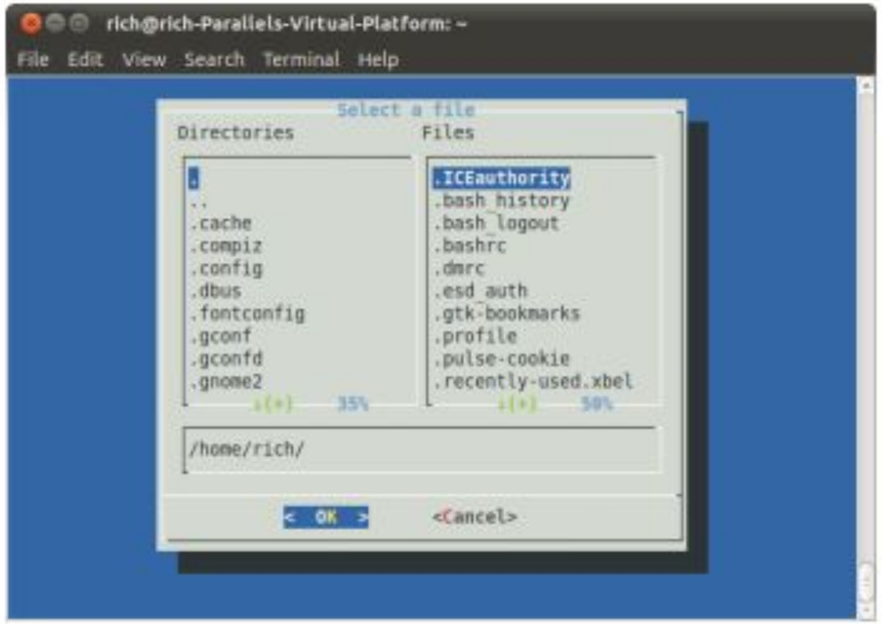
- **The fselect widget:**
  The `fselect` widget format looks like:

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

  The first 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 file listing on the right side that shows all the files in the selected directory, and a simple textbox that contains the currently selected file or directory.

FIGURE 18-7

The fselect widget



- **Using the dialog command in a script:**
  Using the `dialog` command in your scripts is a snap. There are just two things you must remember:
  ■ Check the exit status of the `dialog` command if there's a Cancel or No button available.

  ■ Redirect `STDERR` to retrieve the output value.

```
$ cat menu3
#!/bin/bash
# using dialog to create a menu

temp=$(mktemp -t test.XXXXXX)
temp2=$(mktemp -t test2.XXXXXX)

function diskspace {
   df -k > $temp
   dialog --textbox $temp 20 60
}

function whoseon {
   who > $temp
   dialog --textbox $temp 20 50
}

function memusage {
   cat /proc/meminfo > $temp
   dialog --textbox $temp 20 50
}

while [ 1 ]
do
dialog --menu "Sys Admin Menu" 20 30 10 1 "Display disk space" 2
"Display users" 3 "Display memory usage" 0 "Exit" 2> $temp2
if [ $? -eq 1 ]
then
   break
fi
```

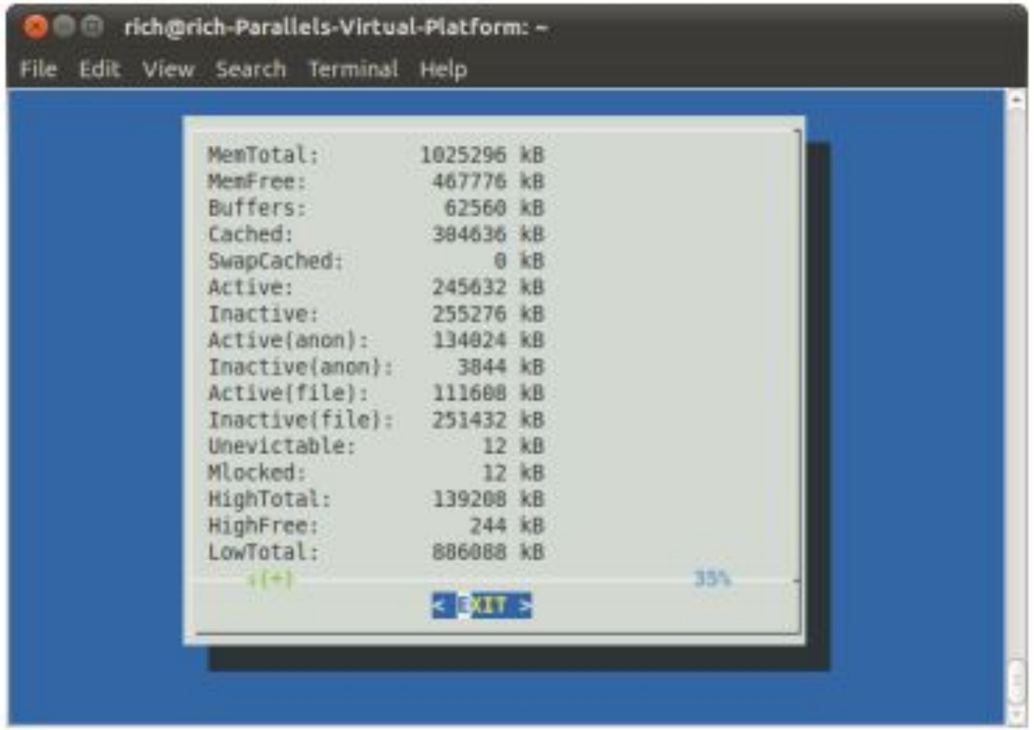
```
selection=$(cat $temp2)

case $selection in
1)
   diskspace ;;
2)
   whoseon ;;
3)
   memusage ;;
0)
   break ;;
*)
   dialog --msgbox "Sorry, invalid selection" 10 30

esac
done
rm -f $temp 2> /dev/null
rm -f $temp2 2> /dev/null
$
```

The `menu` dialog includes a Cancel button, so the script checks the exit status of the dialog command in case the user presses the Cancel button to exit. Because it's in a `while` loop, exiting is as easy as using the `break` command to jump out of the `while` loop.

- The script uses the `mktemp` command to create two temporary files for holding data for the `dialog` commands. The first one, `$temp`, is used to hold the output of the `df`, `whoeson`, and `meminfo` commands so they can be displayed in the `textbox` dialog (see Figure 18-8). The second temporary file, $temp2, is used to hold the selection value from the main menu dialog.

FIGURE 18-8

The `meminfo` command output displayed using the textbox dialog option



Now this is starting to look like a real application that you can show off to people!

- **Getting Graphic:**
  kdialog widgets
  Just like the `dialog` command, the `kdialog` command uses command line options to specify what type of window widget to use. The following is the format of the `kdialog` command:

  ```
  kdialog display-options window-options arguments
  ```

  The `window-options` options allow you to specify what type of window widget to use.

- The `checklist` and `radiolist` widgets allow you to define individual items in the lists and whether they are selected by default:

  ```
  $kdialog --checklist "Items I need" 1 "Toothbrush" on 2 "Toothpaste"
  off 3 "Hair brush" on 4 "Deodorant" off 5 "Slippers" off
  ```

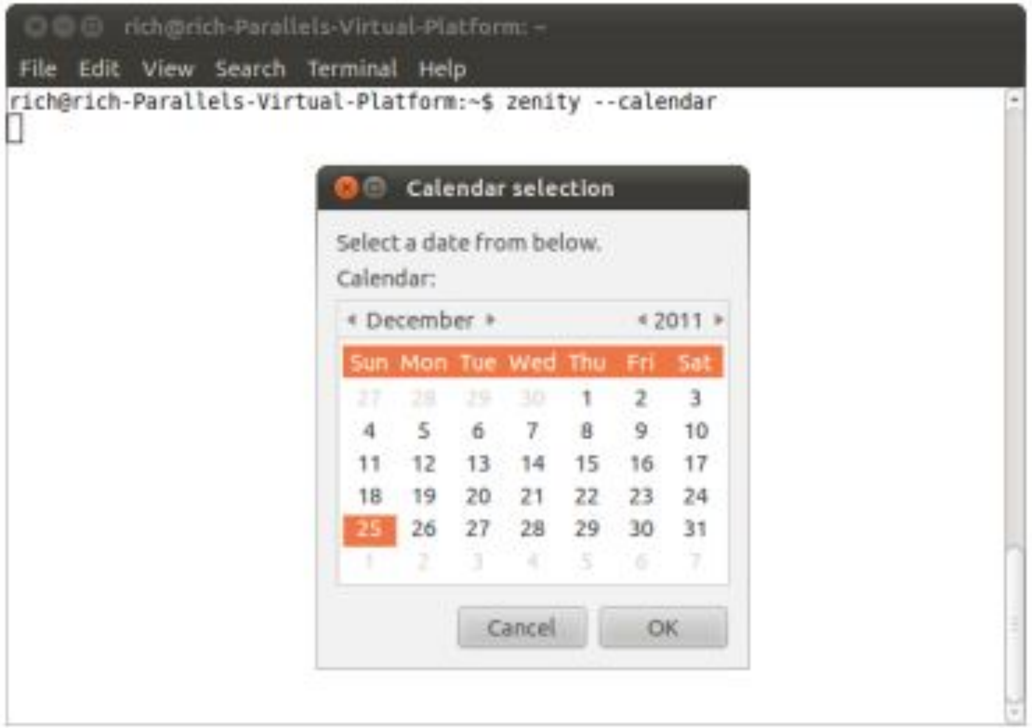
- The items specified as “on” are highlighted in the checklist. To select or deselect an item in the checklist, just click it. If you select the OK button, the `kdialog` sends the tag values to `STDOUT`:

  ```
  "1" "3"
  $
  ```

A kdialog checklist dialog window

FIGURE 18-11

The zenity calendar dialog window



When you select a date from the calendar, the `zenity` command returns the value to `STDOUT`, just like `kdialog`:

```
$ zenity --calendar
12/25/2011
$
```

Introducing sed and gawk

- **Manipulating Text:**

- **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.
  The `sed` editor can manipulate data in a data stream based on commands you either enter into the command line or store in a command text file. The `sed` editor does these things:

    1. Reads one data line at a time from the input
    2. Matches that data with the supplied editor commands
    3. Changes data in the stream as specified in the commands
    4. Outputs the new data to `STDOUT`

- **Defining an editor command in the command line:**
- This example uses the `s` command in the `sed` editor. The `s` command substitutes a second text string for the first text string pattern specified between the forward slashes. In this example, the words `big test` were substituted for the word test.

```
$ echo "This is a test" | sed 's/test/big test/'
This is a big test
$
```

```
$ 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.
$
```

- It's important to note that the `sed` editor doesn't modify the data in the text file itself. It only sends the modified text to `STDOUT`. If you look at the text file, 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

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

- **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 file. Use the -f option to specify the file 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.
$
```

- **Getting to know the gawk program:**
  The `gawk` program takes stream editing one step further than the `sed` editor by providing a programming language instead of just editor commands. Within the `gawk` programming language, you can do the following:

    - ■ Define variables to store data.

      ■ Use arithmetic and string operators to operate on data.

      ■ Use structured programming concepts, such as `if-then` statements and loops, to add logic to your data processing.

      ■ Generate formatted reports by extracting data elements within the data file and repositioning them in another order or format.

  - The `gawk` program's report-generating capabilities are often used for extracting data elements from large bulky text files and formatting them into a readable report.
  - The perfect example of this is formatting log files. Trying to pore through lines of errors in a log file can be difficult.
  - The `gawk` program allows you to filter just the data elements you want to view from the log file, and then you can format them in a manner that makes reading the important data easier.

- If you type a line of text and press the Enter key, `gawk` runs the text through the program script. Just like the `sed` editor, the `gawk` program executes the program script on each line of text available in the data stream. Because the **program script is set to display a fixed text string, no matter what text you enter in the data stream, you get the same text output:**

```
$ gawk '{print "Hello World!"}'
This is a test
Hello World!
hello
Hello World!
This is another test
Hello World!
```

- The Ctrl+D key combination generates an EOF character in bash. Using that key combination terminates
  the `gawk` program and returns you to a command line interface prompt.

- **Using data field variables:**
  It does this by automatically assigning a variable to each data element in a line. By default, `gawk` assigns the following variables to each data field it detects in the line of text:

  - ■ `$0` represents the entire line of text.
    ■ `$1` represents the first data field in the line of text.
    ■ `$2` represents the second data field in the line of text.
    ■ `$n` represents the nth data field in the line of text.

- Here's an example `gawk` program that reads a text file and displays only the first data field value:

```
$ cat data2.txt
One line of test text.
Two lines of test text.
Three lines of test text.
$
$ gawk '{print $1}' data2.txt
One
Two
Three
$
```

- **Using multiple commands in the program script:**
  To use multiple commands in the program script specified on the command line, just place a semicolon between each command:

```
$ echo "My name is Rich" | gawk '{$4="Christine"; print $0}'
My name is Christine
$
```

- The first command assigns a value to the `$4` field variable. The second command then prints the entire data field. Notice from the output that the `gawk` program replaced the fourth data field in the original text with the new value.

- **Running scripts before processing data:**

  By default, `gawk` reads a line of text from the input and then executes the program script on the data

  in the line of text. Sometimes, you may need to run a script before processing data, such as to create a header section for a report. The BEGIN keyword is used to accomplish this. It forces gawk to execute the program script specified after the BEGIN keyword, before gawk reads the data:

```
$ gawk 'BEGIN {print "Hello World!"}'
Hello World!
$
```

  This time the `print` command displays the text before reading any data. However, after it displays the text, it quickly exits, without waiting for any data.

- The reason for this is that the `BEGIN` keyword only applies the specified script before it processes any data. If you want to process data with a normal program script, you must define the program using another script section:

```
$ cat data3.txt
Line 1
Line 2
Line 3
$
$ gawk 'BEGIN {print "The data3 File Contents:"}
> {print $0}' data3.txt
The data3 File Contents:
Line 1
Line 2
Line 3
$
```

  Now after `gawk` executes the `BEGIN` script, it uses the second script to process any file data. Be careful when doing this; both of the scripts are still considered one text string on the `gawk` command line. You need to place your single quotation marks accordingly.

- **Running scripts after processing data**
  Like the `BEGIN` keyword, the `END` keyword allows you to specify a program script that `gawk` executes after reading the data:

```
$ gawk 'BEGIN {print "The data3 File Contents:"}
> {print $0}
> END {print "End of File"}' data3.txt
The data3 File Contents:
Line 1
Line 2
Line 3
End of File
$
```

  When the `gawk` program is finished printing the file contents, it executes the commands in the `END` script. This is a great technique to use to add footer data to reports after all the normal data has been processed.

- **Commanding at the sed Editor Basics:**

  Introducing more substitution options :

```
$ 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.
$
```

- Four types of substitution flags are available:
  - ■ 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.

The w substitution flag produces the same output but stores the output in the specified file:

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

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 specific 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 filters out a line

Both forms use the same format for specifying the address:

`[`*address*`]command`

- **Addressing the numeric 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 sed command 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 sed editor modified the text only in line two per the address specified. 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
$
```

- **Deleting lines:**
  The text substitution command isn't the only command available in the `sed` editor. If you need to delete specific lines of text in a text stream, you can use the `delete` command.

```
$ cat data6.txt
This is line number 1.
This is line number 2.
This is line number 3.
```

```
This is line number 4.
$
$ sed '3d' data6.txt
This is line number 1.
This is line number 2.
This is line number 4.
$
```

or by a specific range of lines:

```
$ sed '2,3d' data6.txt
This is line number 1.
This is line number 4.
$
```

or by using the special end-of-file character:

```
$ sed '3,$d' data6.txt
This is line number 1.
This is line number 2.
$
```

The pattern-matching feature of the `sed` editor also applies to the `delete` command:

```
$ sed '/number 1/d' data6.txt
This is line number 2.
This is line number 3.
This is line number 4.
$
```

• Inserting and appending text:

- As you would expect, like any other editor, the `sed` editor allows you to insert and append text lines to the data stream. The difference between the two actions can be confusing:
  - ■ The `insert` command (`i`) adds a new line before the specified line.
  - ■ The `append` command (`a`) adds a new line after the specified line.

The text in *`new line`* appears in the `sed` editor output in the place you specify. Remember that when you use the `insert` command, the **text appears before the data stream text:**

```
$ echo "Test Line 2" | sed 'i\Test Line 1'
Test Line 1
Test Line 2

$
```

And when you use the `append` command, the **text appears after the data stream text:**

```
$ echo "Test Line 2" | sed 'a\Test Line 1'
Test Line 2
Test Line 1

$
```

Here's an example of inserting a new line before line 3 in the data stream:

```
$ sed '3i\
> This is an inserted line.' data6.txt
This is line number 1.
This is line number 2.
This is an inserted line.
This is line number 3.
This is line number 4.
$
```

Here's an example of appending a new line after line 3 in the data stream:

```
$ sed '3a\
> This is an appended line.' data6.txt
This is line number 1.
This is line number 2.
This is line number 3.
This is an appended line.
This is line number 4.
$
```

- you want to append a new line of text to the end of a data stream, just use the dollar sign, which represents the last line of data:

```
$ sed '$a\
> This is a new line of text.' data6.txt
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
This is a new line of text.
$
```

To insert or append more than one line of text, you must use a backslash on each line of new text until you reach the last text line where you want to insert or append text:

```
$ sed '1i\
> This is one line of new text.\
> This is another line of new text.' data6.txt
This is one line of new text.
This is another line of new text.
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
$
```

- **Changing lines**
  The `change` command allows you to change the contents of an entire line of text in the data stream. It works the same way as the `insert` and `append` commands, in that you must specify the new line separately from the rest of the sed command:

```
$ sed '3c\
> This is a changed line of text.' data6.txt
This is line number 1.
This is line number 2.
This is a changed line of text.
This is line number 4.
$
```

In this example, the sed editor changes the text in line number 3. You can also use a text pattern for the address:

```
$ sed '/number 3/c\
> This is a changed line of text.' data6.txt
This is line number 1.
This is line number 2.
This is a changed line of text.
This is line number 4.
$
```

The text pattern change command changes any line of text in the data stream that it matches.

```
$ cat data8.txt
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
This is line number 1 again.
This is yet another line.
This is the last line in the file.
$
$ sed '/number 1/c\
> This is a changed line of text.' data8.txt
This is a changed line of text.
This is line number 2.
This is line number 3.
This is line number 4.
This is a changed line of text.
This is yet another line.
This is the last line in the file.
$
```

- **Transforming characters:**
  The `transform` command performs a one-to-one mapping of the `inchars` and the `outchars` values. The first character in `inchars` is converted to the first character in `outchars`. The second character in `inchars` is converted to the second character in `outchars`. This mapping continues throughout the length of the specified characters. If the `inchars` and `outchars` are not the same length, the sed editor produces an error message.

Here's a simple example of using the `transform` command:

```
$ sed 'y/123/789/' data8.txt
This is line number 7.
This is line number 8.
This is line number 9.
This is line number 4.
This is line number 7 again.
This is yet another line.
This is the last line in the file.
$
```

- **Printing revisited:**
  In addition, three commands that can be used to print information from the data stream:
  ■ The `p` command to print a text line
  ■ The equal sign (=) command to print line numbers
  ■ The `l` (lowercase L) command to list a line

- All it does is print the data text that you already know is there. The most common use for the `print` command is printing lines that contain matching text from a text pattern:

```
$ cat data6.txt
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
$
$ sed -n '/number 3/p' data6.txt
This is line number 3.
$
```

By using the `-n` option on the command line, you can suppress all the other lines and print only the line that contains the matching text pattern.

You can also use this as a quick way to print a subset of lines in a data stream:

```
$ sed -n '2,3p' data6.txt
This is line number 2.
This is line number 3.
$
```

- **Printing line numbers:** The `equal sign` command prints the current line number for the line within the data stream.

```
$ 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 '=' data1.txt
1
The quick brown fox jumps over the lazy dog.
2
The quick brown fox jumps over the lazy dog.
3
The quick brown fox jumps over the lazy dog.
4
The quick brown fox jumps over the lazy dog.
$
```

• **Using fi les with sed**

The `substitution` command contains flags that allow you to work with files. There are also regular `sed` editor commands that let you do that without having to substitute text. Writing to a file The `w` command is used to write lines to a file. Here's the format for the `w` command:

[*address*]w *filename*

Here's an example that prints only the first two lines of a data stream to a text file:

```
$ sed '1,2w test.txt' data6.txt
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
$
$ cat test.txt
This is line number 1.
This is line number 2.
$
```

- **Reading data from a file**
  You've already seen how to insert data into and append text to a data stream from the `sed` command line. The `read` command (`r`) allows you to insert data contained in a separate file. Here's the format of the `read` command:

  `[`*`address`*`]r `*`filename`*

- The *`filename`* parameter specifies either an absolute or relative pathname for the file that contains the data. You can't use a range of addresses for the `read` command. You can only specify a single line number or text pattern address. The sed editor inserts the text from the file after the address

```
$ cat data12.txt
This is an added line.
This is the second added line.
$
$ sed '3r data12.txt' data6.txt
This is line number 1.
This is line number 2.
This is line number 3.
This is an added line.
This is the second added line.
This is line number 4.
$
```

If you want to add text to the end of a data stream, just use the dollar sign address symbol:

```
$ sed '$r data12.txt' data6.txt
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
This is an added line.
This is the second added line.
$
```