

28

READING KEYBOARD INPUT



The scripts we have written so far lack a feature common in most computer programs : *interactivity*, or the capability of the program to interact with the user. While many programs don't need to be interactive, some programs benefit from being able to accept input directly from the user. Take, for example, this script from the previous chapter:

```
#!/bin/bash
```

```
# test-integer2: evaluate the value of an integer.
```

```
INT=-5
```

```
if [[ "$INT" =~ ^-[0-9]+$ ]]; then
```

```
    if [ "$INT" -eq 0 ]; then
```

```
        echo "INT is zero."
```

```
    else
```

```
        if [ "$INT" -lt 0 ]; then
```

```
            echo "INT is negative."
```

```
        else
```

```
            echo "INT is positive."
```

```
        fi
```

```
    if [ $((INT % 2)) -eq 0 ]; then
```

```
        echo "INT is even."
```

```
        else
            echo "INT is odd."
        fi
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

Each time we want to change the value of `INT`, we have to edit the script. It would be much more useful if the script could ask the user for a value. In this chapter, we will begin to look at how we can add interactivity to our programs.

read—Read Values from Standard Input

The `read` builtin command is used to read a single line of standard input. This command can be used to read keyboard input or, when redirection is employed, a line of data from a file. The command has the following syntax:

```
read [-options] [variable...]
```

where *options* is one or more of the available options listed later in Table 28-1 and *variable* is the name of one or more variables used to hold the input value. If no variable name is supplied, the shell variable `REPLY` contains the line of data.

Basically, `read` assigns fields from standard input to the specified variables. If we modify our integer evaluation script to use `read`, it might look like this:

```
#!/bin/bash
```

```
# read-integer: evaluate the value of an integer.
```

```
echo -n "Please enter an integer -> "
```

```
read int
```

```
if [[ "$int" =~ ^-[0-9]+$ ]]; then
    if [ "$int" -eq 0 ]; then
        echo "$int is zero."
    else
        if [ "$int" -lt 0 ]; then
            echo "$int is negative."
        else
            echo "$int is positive."
        fi
        if [ $((int % 2)) -eq 0 ]; then
            echo "$int is even."
        else
            echo "$int is odd."
        fi
    fi
else
    echo "Input value is not an integer." >&2
    exit 1
fi
```

We use `echo` with the `-n` option (which suppresses the trailing newline on output) to display a prompt, and then we use `read` to input a value for the variable `int`. Running this script results in this:

```
[me@linuxbox ~]$ read-integer
```

```
Please enter an integer -> 5
```

```
5 is positive.
```

```
5 is odd.
```

`read` can assign input to multiple variables, as shown in this script:

```
#!/bin/bash
```

```
# read-multiple: read multiple values from keyboard
```

```
echo -n "Enter one or more values > "  
read var1 var2 var3 var4 var5
```

```
echo "var1 = '$var1'"  
echo "var2 = '$var2'"  
echo "var3 = '$var3'"  
echo "var4 = '$var4'"  
echo "var5 = '$var5'"
```

In this script, we assign and display up to five values. Notice how `read` behaves when given different numbers of values, shown here:

```
[me@linuxbox ~]$ read-multiple
```

```
Enter one or more values > a b c d e
```

```
var1 = 'a'
```

```
var2 = 'b'
```

```
var3 = 'c'
```

```
var4 = 'd'
```

```
var5 = 'e'
```

```
[me@linuxbox ~]$ read-multiple
```

```
Enter one or more values > a
```

```
var1 = 'a'
```

```
var2 = ''
```

```
var3 = ''
```

```
var4 = ''
```

```
var5 = ''
```

```
[me@linuxbox ~]$ read-multiple
```

```
Enter one or more values > a b c d e f g
```

```
var1 = 'a'
```

```
var2 = 'b'
```

```
var3 = 'c'
```

```
var4 = 'd'
```

```
var5 = 'e f g'
```

If `read` receives fewer than the expected number, the extra variables are empty, while an excessive amount of input results in the final variable containing all of the extra input.

If no variables are listed after the `read` command, a shell variable, `REPLY`, will be assigned all the input.

```
#!/bin/bash
```

```
# read-single: read multiple values into default variable
```

```
echo -n "Enter one or more values > "  
read
```

```
echo "REPLY = '$REPLY'"
```

Running this script results in this:

```
[me@linuxbox ~]$ read-single  
Enter one or more values > a b c d  
REPLY = 'a b c d'
```

Options

`read` supports the options described in [Table 28-1](#).

Table 28-1: `read` Options

Option	Description
--------	-------------

<code>-a array</code>	Assign the input to <i>array</i> , starting with index zero. We will cover arrays in Chapter 35 .
-----------------------	---

<code>-d delimiter</code>	The first character in the string <i>delimiter</i> is used to indicate the end of input, rather than a newline character.
---------------------------	---

Option	Description
--------	-------------

-e	Use Readline to handle input. This permits input editing in the same manner as the command line.
-i <i>string</i>	Use <i>string</i> as a default reply if the user simply presses ENTER. Requires the -e option.
-n <i>num</i>	Read <i>num</i> characters of input, rather than an entire line.
-p <i>prompt</i>	Display a prompt for input using the string <i>prompt</i> .
-r	Raw mode. Do not interpret backslash characters as escapes.
-s	Silent mode. Do not echo characters to the display as they are typed. This is useful when inputting passwords and other confidential information.
-t <i>seconds</i>	Timeout. Terminate input after <i>seconds</i> . <code>read</code> returns a non-zero exit status if an input times out.
-u <i>fd</i>	Use input from file descriptor <i>fd</i> , rather than standard input.

Using the various options, we can do interesting things with `read`. For example, with the `-p` option, we can provide a prompt string.

```
#!/bin/bash
```

```
# read-single: read multiple values into default variable
```

```
read -p "Enter one or more values > "
```

```
echo "REPLY = '$REPLY'"
```

With the `-t` and `-s` options, we can write a script that reads “secret” input and times out if the input is not completed in a specified time.

```
#!/bin/bash

# read-secret: input a secret passphrase

if read -t 10 -sp "Enter secret passphrase > " secret_pass; then
    echo -e "\nSecret passphrase = '$secret_pass'"
else
    echo -e "\nInput timed out" >&2
    exit 1
fi
```

The script prompts the user for a secret passphrase and waits ten seconds for input. If the entry is not completed within the specified time, the script exits with an error. Because the `-s` option is included, the characters of the passphrase are not echoed to the display as they are typed.

It’s possible to supply the user with a default response using the `-e` and `-i` options together.

```
#!/bin/bash

# read-default: supply a default value if user presses Enter key.

read -e -p "What is your user name? " -i $USER
echo "You answered: '$REPLY'"
```

In this script, we prompt the user to enter a username and use the environment variable `USER` to provide a default value. When the script is run, it displays the default string, and if the user simply presses `ENTER`, `read` will assign the default string to the `REPLY` variable.

```
[me@linuxbox ~]$ read-default
What is your user name? me
```

IFS

Normally, the shell performs word-splitting on the input provided to `read`. As we have seen, this means that multiple words separated by one or more spaces become separate items on the input line and are assigned to separate variables by `read`. This behavior is configured by a shell variable named `IFS` (for Internal Field Separator). The default value of `IFS` contains a space, a tab, and a newline character, each of which will separate items from one another.

We can adjust the value of `IFS` to control the separation of fields input to `read`. For example, the `/etc/passwd` file contains lines of data that use the colon character as a field separator. By changing the value of `IFS` to a single colon, we can use `read` to input the contents of `/etc/passwd` and successfully separate fields into different variables. Here we have a script that does just that:

```
#!/bin/bash
```

```
# read-ifs: read fields from a file
```

```
FILE=/etc/passwd
```

```
read -p "Enter a username > " user_name
```

```
file_info="$(grep "^$user_name:" $FILE)"
```

```
if [ -n "$file_info" ]; then
```

```
    IFS=":" read user pw uid gid name home shell <<< "$file_info"
```

```
    echo "User =    '$user'"
```

```
    echo "UID =     '$uid'"
```

```
    echo "GID =     '$gid'"
```

```
    echo "Full Name = '$name'"
```

```
    echo "Home Dir. = '$home'"
```



```
    echo "Shell =   '$shell'"
else
    echo "No such user '$user_name'" >&2
    exit 1
fi
```

This script prompts the user to enter the username of an account on the system and then displays the different fields found in the user's record in the */etc/passwd* file. The script contains two interesting lines. The first is as follows:

```
file_info=$(grep "^$user_name:" $FILE)
```

This line assigns the results of a `grep` command to the variable `file_info`. The regular expression used by `grep` assures that the username will match only a single line in the */etc/passwd* file.

The second interesting line is this one:

```
IFS=":" read user pw uid gid name home shell <<< "$file_info"
```

The line consists of three parts: a variable assignment, a `read` command with a list of variable names as arguments, and a strange new redirection operator. We'll look at the variable assignment first.

The shell allows one or more variable assignments to take place immediately before a command. These assignments alter the environment for the command that follows. The effect of the assignment is temporary, changing the environment only for the duration of the command. In our case, the value of `IFS` is changed to a colon character. Alternately, we could have coded it this way:

```
OLD_IFS="$IFS"
IFS=":"
read user pw uid gid name home shell <<< "$file_info"
IFS="$OLD_IFS"
```

where we store the value of `IFS`, assign a new value, perform the `read` command, and then restore `IFS` to its original value. Clearly, placing the variable assignment in front of the command is a more concise way of doing the same thing.

The `<<<` operator indicates a *here string*. A here string is like a here document, only shorter, consisting of a single string. In our example, the line of data from the `/etc/passwd` file is fed to the standard input of the `read` command. We might wonder why this rather oblique method was chosen rather than this:

```
echo "$file_info" | IFS=":" read user pw uid gid name home shell
```

Well, there's a reason . . .

YOU CAN'T PIPE READ

While the `read` command normally takes input from standard input, you cannot do this:

```
echo "foo" | read
```

We would expect this to work, but it does not. The command will appear to succeed, but the `REPLY` variable will always be empty. Why is this?

The explanation has to do with the way the shell handles pipelines. In `bash` (and other shells such as `sh`), pipelines create *subshells*. These are copies of the shell and its environment that are used to execute the command in the pipeline. In the previous example, `read` is executed in a subshell.

Subshells in Unix-like systems create copies of the environment for the processes to use while they execute. When the processes finishes, the copy of the environment is destroyed. This means that *a subshell can never alter the environment of its parent process*. `read` assigns variables, which then become part of the environment. In the previous example,

`read` assigns the value `foo` to the variable `REPLY` in its subshell's environment, but when the command exits, the subshell and its environment are destroyed, and the effect of the assignment is lost.

Using here strings is one way to work around this behavior. Another method is discussed in [Chapter 36](#).

Validating Input

With our new ability to have keyboard input comes an additional programming challenge: validating input. Often the difference between a well-written program and a poorly written one lies in the program's ability to deal with the unexpected. Frequently, the unexpected appears in the form of bad input. We've done a little of this with our evaluation programs in the previous chapter, where we checked the values of integers and screened out empty values and non-numeric characters. It is important to perform these kinds of programming checks every time a program receives input to guard against invalid data. This is especially important for programs that are shared by multiple users. Omitting these safeguards in the interests of economy might be excused if a program is to be used once and only by the author to perform some special task. Even then, if the program performs dangerous tasks such as deleting files, it would be wise to include data validation, just in case.

Here we have an example program that validates various kinds of input:

```
#!/bin/bash

# read-validate: validate input

invalid_input () {
    echo "Invalid input '$REPLY'" >&2
    exit 1
}
```

```
read -p "Enter a single item > "

# input is empty (invalid)
[[ -z "$REPLY" ]] && invalid_input

# input is multiple items (invalid)
(( "$(echo "$REPLY" | wc -w)" > 1 )) && invalid_input

# is input a valid filename?
if [[ "$REPLY" =~ ^[-[:alnum:]\.]+$ ]]; then
    echo "'$REPLY' is a valid filename."
    if [[ -e "$REPLY" ]]; then
        echo "And file '$REPLY' exists."
    else
        echo "However, file '$REPLY' does not exist."
    fi
fi

# is input a floating point number?
if [[ "$REPLY" =~ ^-?[:digit:]*\.[[:digit:]]+$ ]]; then
    echo "'$REPLY' is a floating point number."
else
    echo "'$REPLY' is not a floating point number."
fi

# is input an integer?
if [[ "$REPLY" =~ ^-?[:digit:]+$ ]]; then
    echo "'$REPLY' is an integer."
else
    echo "'$REPLY' is not an integer."
fi
else
    echo "The string '$REPLY' is not a valid filename."
fi
```

This script prompts the user to enter an item. The item is subsequently analyzed to determine its contents. As we can see, the script makes use of many of the concepts that we have covered thus far, including shell functions, `[]`, `(())`, the control operator `&&`, and `if`, as well as a healthy dose of regular expressions.

Menus

A common type of interactivity is called *menu-driven*. In menu-driven programs, the user is presented with a list of choices and is asked to choose one. For example, we could imagine a program that presented the following:

Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit

Enter selection [0-3] >

Using what we learned from writing our `sys_info_page` program, we can construct a menu-driven program to perform the tasks on the previous menu.

```
#!/bin/bash
```

```
# read-menu: a menu driven system information program
```

```
clear
```

```
echo "
```

```
Please Select:
```

1. Display System Information
2. Display Disk Space

3. Display Home Space Utilization

0. Quit

"

```
read -p "Enter selection [0-3] > "
```

```
if [[ "$REPLY" =~ ^[0-3]$ ]]; then
```

```
    if [[ "$REPLY" == 0 ]]; then
```

```
        echo "Program terminated."
```

```
        exit
```

```
    fi
```

```
    if [[ "$REPLY" == 1 ]]; then
```

```
        echo "Hostname: $HOSTNAME"
```

```
        uptime
```

```
        exit
```

```
    fi
```

```
    if [[ "$REPLY" == 2 ]]; then
```

```
        df -h
```

```
        exit
```

```
    fi
```

```
    if [[ "$REPLY" == 3 ]]; then
```

```
        if [[ "$(id -u)" -eq 0 ]]; then
```

```
            echo "Home Space Utilization (All Users)"
```

```
            du -sh /home/*
```

```
        else
```

```
            echo "Home Space Utilization ($USER)"
```

```
            du -sh "$HOME"
```

```
        fi
```

```
        exit
```

```
    fi
```

```
else
```

```
    echo "Invalid entry." >&2
```

```
    exit 1
```

```
fi
```

This script is logically divided into two parts. The first part displays the menu and inputs the response from the user. The second part identifies the response and carries out the selected action. Notice the use of the `exit` command in this script. It is used here to prevent the script from executing unnecessary code after an action has been carried out. The presence of multiple exit points in a program is generally a bad idea (it makes program logic harder to understand), but it works in this script.

Summing Up

In this chapter, we took our first steps toward interactivity, allowing users to input data into our programs via the keyboard. Using the techniques presented thus far, it is possible to write many useful programs, such as specialized calculation programs and easy-to-use front ends for arcane command line tools. In the next chapter, we will build on the menu-driven program concept to make it even better.

Extra Credit

It is important to study the programs in this chapter carefully and have a complete understanding of the way they are logically structured, as the programs to come will be increasingly complex. As an exercise, rewrite the programs in this chapter using the `test` command rather than the `[[]]` compound command. Hint: Use `grep` to evaluate the regular expressions and evaluate the exit status. This will be good practice.