Leyendo entrada de tecladol.

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:
Los guiones que hemos escrito hasta ahora carecen de una característica común en la mayoría de los programas de computadora: la interactividad o la capacidad del programa para interactuar con el usuario. Si bien muchos programas no necesitan ser interactivos, algunos programas se benefician de poder aceptar información directamente del usuario. Tomemos, por ejemplo, este script del capítulo anterior:

```
#!/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.
Cada vez que queremos cambiar el valor de INT, tenemos que editar el script. Sería mucho más útil si el script pudiera pedirle al usuario un valor. En este capítulo, comenzaremos a ver cómo podemos agregar interactividad a nuestros programas.

read -- Read Values from Standard Input
read - Leer valores de entrada estándar

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:
El comando read incorporado se usa para leer una sola línea de entrada estándar. Este comando se puede usar para leer la entrada del teclado o, cuando se emplea la redirección, una línea de datos de un archivo. El comando tiene la siguiente sintaxis:

```
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.
donde opciones es una o más de las opciones disponibles que se enumeran más adelante en la Tabla 28-1 y variable es el nombre de una o más variables utilizadas para contener el valor de entrada. Si no se proporciona un nombre de variable, la variable de shell REPLY contiene la línea de datos.

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:
Básicamente, la lectura asigna campos de la entrada estándar a las variables especificadas. Si modificamos nuestro script de evaluación de enteros para usar read, podría verse así:

```
#!/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:
Usamos echo con la opción -n (que suprime el salto de línea final en la salida) para mostrar un mensaje, y luego usamos read para ingresar un valor para la variable int. La ejecución de este script da como resultado esto:

```
[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:
read puede asignar entrada a múltiples variables, como se muestra en este 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:
En este script, asignamos y mostramos hasta cinco valores. Observe cómo se comporta la lectura cuando se le dan diferentes números de valores, que se muestran aquí:

```
[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.
Si la lectura recibe menos del número esperado, las variables adicionales están vacías, mientras que una cantidad excesiva de entrada da como resultado que la variable final contenga toda la entrada adicional.

If no variables are listed after the read command, a shell variable, REPLY, will be assigned all the input.
Si no se enumeran variables después del comando de lectura, se asignará toda la entrada a una variable de shell, REPLY.

```
#!/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:
La ejecución de este script da como resultado esto:

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

Options
Opciones

read supports the options described in Table 28-1.
read admite las opciones descritas en la Tabla 28-1.

Table 28-1: read Options
Tabla 28-1: opciones de read

Option	Description
-a array	Assign the input to array , starting with index zero. We will cover arrays in Chapter 35. Asigne la entrada a la matriz, comenzando con el índice cero. Cubriremos las matrices en el Capítulo 35.
-d delimiter	The first character in the string delimiter is used to indicate the end of input, rather than a newline character. El primer carácter del delimitador de cadena se utiliza para indicar el final de la entrada, en lugar de un carácter de nueva línea.
-e	Use Readline to handle input. This permits input editing in the same manner as the command line. Utilice Readline para manejar la entrada. Esto permite la edición de entradas de la misma manera que la línea de comando.
-i string	Use string as a default reply if the user simply presses the -e option. Utilice una cadena como respuesta predeterminada si el usuario simplemente presiona la opción -e.
-n num	Read num characters of input, rather than an entire line. Leer num caracteres de entrada, en lugar de una línea completa.
-p prompt	Display a prompt for input using the string prompt. Muestre una solicitud de entrada utilizando la solicitud de cadena.
-r	Raw mode. Do not interpret backslash characters as escapes. Modo crudo. No interprete los caracteres de barra invertida como 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. Modo silencioso. No haga eco de caracteres en la pantalla mientras se escriben. Esto es útil al ingresar contraseñas y otra información confidencial.
-t seconds	Timeout. Terminate input after seconds . read returns a non-zero exit status if an input times out. Se acabó el tiempo. Termine la entrada después de segundos. read devuelve un estado de salida distinto de cero si se agota el tiempo de espera de una entrada.
-u fd	Use input from file descriptor fd , rather than standard input. Utilice la entrada del descriptor de archivo fd, en lugar de la entrada estándar.

Using the various options, we can do interesting things with read . For example, with the -p option, we can provide a prompt string.
Usando las distintas opciones, podemos hacer cosas interesantes con read. Por ejemplo, con la opción -p, podemos proporcionar una cadena de solicitud.

```
#!/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.
Con las opciones -t y -s, podemos escribir un script que lea la entrada “secreta” y se agote si la entrada no se completa en un tiempo especificado.

```
#!/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.
El script solicita al usuario una frase de contraseña secreta y espera diez segundos para ingresar. Si la entrada no se completa dentro del tiempo especificado, el script sale con un error. Debido a que se incluye la opción -s, los caracteres de la frase de contraseña no se repiten en la pantalla a medida que se escriben. Es posible proporcionar al usuario una respuesta predeterminada utilizando las opciones -e y -i juntas.

```
#!/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.
En este script, solicitamos al usuario que ingrese un nombre de usuario y usamos la variable de entorno USER para proporcionar un valor predeterminado. Cuando se ejecuta el script, muestra la cadena predeterminada, y si el usuario simplemente presiona enter, read asignará la cadena predeterminada a la variable REPLY.

```
[me@linuxbox ~]$ read-default
What is your user name? me
You answered: '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.
Normalmente, el shell realiza la división de palabras en la entrada proporcionada para leer. Como hemos visto, esto significa que varias palabras separadas por uno o más espacios se convierten en elementos separados en la línea de entrada y se asignan a variables separadas por lectura. Este comportamiento se configura mediante una variable de shell denominada IFS (para Internal Field Separator). El valor predeterminado de IFS contiene un espacio, una pestaña y un carácter de nueva línea, cada uno de los cuales separará los elementos entre sí.

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:
Podemos ajustar el valor de IFS para controlar la separación de los campos de entrada a leer. Por ejemplo, el archivo / etc / passwd contiene líneas de datos que usan el carácter de dos puntos como separador de campo. Al cambiar el valor de IFS a dos puntos simples, podemos usar read para ingresar el contenido de / etc / passwd y separar con éxito los campos en diferentes variables. Aquí tenemos un script que hace precisamente eso:

```bash
#!/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 differen
```

```bash
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 gre
The second interesting line is this one:<br /><span style="color:green">La segunda línea interesante es esta:
```

```bash
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 argu
The shell allows one or more variable assignments to take place immediately before a command. These assignmen
```

bash
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 o
The <<< operator indicates a here string. A here string is like a here document, only shorter, consisting of
```

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

```
Well, there's a reason . . .<br /><span style="color:green">Bueno, hay una razón. . .</span>



> You Can't Pipe read<br /><span style="color:green">no puedes tubear lectura</span><br />
While the read command normally takes input from standard input, you cannot do this:<br /><span style="color:

> ` echo "foo" | read `

> We would expect this to work, but it does not. The command will appear to succeed, but the REPLY variable w
The explanation has to do with the way the shell handles pipelines. In bash (and other shells such as sh ), p
Subshells in Unix-like systems create copies of the environment for the processes to use while they execute.
Using here strings is one way to work around this behavior. Another method is discussed in Chapter 36.<br /><


# Validating Input<br /><span style="color:yellow">Validando entrada </span>

With our new ability to have keyboard input comes an additional programming challenge: validating input. Ofte

Here we have an example program that validates various kinds of input:<br /><span style="color:green">Aquí te
_______
```

bash
#!/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

```
# 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. A


# Menus<br /><span style="color:yellow">Menús</span>

A common type of interactivity is called menu-driven. In menu-driven programs, the user is presented with a l
_________
```

shell
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
4. Quit
   Enter selection [0-3] >

```
__________
Using what we learned from writing our sys_info_page program, we can construct a menu-driven program to perfo
______
```

bash
#!/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
    4. 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.
Este script está lógicamente dividido en dos partes. La primera parte muestra el menú e ingresa la respuesta del usuario. La segunda parte identifica la respuesta y realiza la acción seleccionada. Observe el uso del comando de salida en este script. Se utiliza aquí para evitar que el script ejecute código innecesario después de que se haya realizado una acción. La presencia de múltiples puntos de salida en un programa es generalmente una mala idea (hace que la lógica del programa sea más difícil de entender), pero funciona en este script.

Summing Up
Resumiendo.

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.
En este capítulo, dimos nuestros primeros pasos hacia la interactividad, permitiendo a los usuarios ingresar datos en nuestros programas a través del teclado. Usando las técnicas presentadas hasta ahora, es posible escribir muchos programas útiles, como programas de cálculo especializados y interfaces fáciles de usar para herramientas de línea de comandos arcanas. En el siguiente capítulo, nos basaremos en el concepto de programa basado en menús para hacerlo aún mejor.

Extra Credit
Crédito adicional

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.
Es importante estudiar detenidamente los programas de este capítulo y tener una comprensión completa de la forma en que están estructurados lógicamente, ya que los programas futuros serán cada vez más complejos. Como ejercicio, vuelva a escribir los programas de este capítulo utilizando el comando test en lugar del comando compuesto [[]]. Sugerencia: use grep para evaluar las expresiones regulares y evaluar el estado de salida. Esta será una buena práctica