32

Positional Parameters
Parámetros posicionales

One feature that has been missing from our programs so far is the ability to accept and process command line options and arguments. In this chapter, we will examine the shell features that allow our programs to get access to the contents of the command line.
Una característica que ha faltado en nuestros programas hasta ahora es la capacidad de aceptar y procesar opciones y argumentos de la línea de comandos. En este capítulo, examinaremos las características del shell que permiten a nuestros programas acceder al contenido de la línea de comandos.

Accessing the Command Line

The shell provides a set of variables called positional parameters that contain the individual words on the command line. The variables are named 0 through 9 . They can be demonstrated this way:

```
#!/bin/bash
# posit-param: script to view command line parameters
echo "
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

This is a simple script that displays the values of the variables $0 – $9 .
Este es un script simple que muestra los valores de las variables $ 0 - $ 9. When executed with no command line arguments, the result is this:
Cuando se ejecuta sin argumentos de línea de comando, el resultado es este:

```
[me@linuxbox ~]$ posit-param
$0 = /home/me/bin/posit-param
$1 =
$2 =
```

```
$3 =
$4 =
$5 =
$6 =
$7 =
$8 =
$9 =
```

Even when no arguments are provided, $0 will always contain the first item appearing on the command line, which is the pathname of the program being executed. When arguments are provided, we see these results:
Incluso cuando no se proporcionan argumentos, $0 siempre contendrá el primer elemento que aparece en la línea de comando, que es la ruta del programa que se está ejecutando. Cuando se proporcionan argumentos, vemos estos resultados:

```
[me@linuxbox ~]$ posit-param a b c d
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

> ## Note
> ## Nota

> You can actually access more than nine parameters using parameter expansion. To specify a number greater than nine, surround the number in braces, as in ${10}, ${55}, ${211}, and so on.
> De hecho, puede acceder a más de nueve parámetros mediante la expansión de parámetros. Para especificar un número mayor que nueve, rodee el número entre llaves, como en $ {10}, $ {55}, $ {211}, etc.

Determining the Number of Arguments
Determiando el número de argumentos

The shell also provides a variable, $# , that contains the number of arguments on the command line.
El shell también proporciona una variable, $ #, que contiene el número de argumentos en la línea de

comando.

```
#!/bin/bash

# posit-param: script to view command line parameters

echo "
Number of arguments: $#
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

This is the result:
Este es el resultado:

```
[me@linuxbox ~]$ posit-param a b c d

Number of arguments: 4
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

shift—Getting Access to Many Arguments
shift - Acceso a muchos argumentos

But what happens when we give the program a large number of arguments such as the following?
Pero, ¿qué sucede cuando le damos al programa una gran cantidad de argumentos como los siguientes?

```
[me@linuxbox ~]$ posit-param *
Number of arguments: 82

$0 = /home/me/bin/posit-param
$1 = addresses.ldif
$2 = bin
$3 = bookmarks.html
$4 = debian-500-i386-netinst.iso
$5 = debian-500-i386-netinst.jigdo
$6 = debian-500-i386-netinst.template
$7 = debian-cd_info.tar.gz
$8 = Desktop
$9 = dirlist-bin.txt
```

On this example system, the wildcard * expands into 82 arguments. How can we process that many? The shell provides a method, albeit a clumsy one, to do this. The shift command causes all the parameters to "move down one" each time it is executed. In fact, by using shift , it is possible to get by with only one parameter (in addition to $0 , which never changes).
En este sistema de ejemplo, el comodín * se expande a 82 argumentos. ¿Cómo podemos procesar tantos? El caparazón proporciona un método, aunque torpe, para hacer esto. El comando de cambio hace que todos los parámetros "bajen uno" cada vez que se ejecuta. De hecho, **al usar shift, es posible arreglárselas con un solo parámetro** (además de $0, que nunca cambia).

```
#!/bin/bash

# posit-param2: script to display all arguments

count=1

while [[ $# -gt 0 ]]; do
    echo "Argument $count = $1"
    count=$((count + 1))
    shift
done
```

Each time shift is executed, the value of $2 is moved to $1 , the value of $3 is moved to $2 , and so on. The value of $# is also reduced by one.
Cada vez que se ejecuta un turno, el valor de $2 se mueve a $1, el valor de $3 se mueve a $2, y así sucesivamente. El valor de $# también se reduce en uno.

In the posit-param2 program, we create a loop that evaluates the number of arguments remaining and continues as long as there is at least one. We display the current argument, increment the variable count with each iteration of the loop to provide a running count of the number of arguments processed, and,

finally, execute a shift to load $1 with the next argument. Here is the program at work:
En el programa posit-param2, creamos un ciclo que evalúa el número de argumentos restantes y continúa mientras haya al menos uno. Mostramos el argumento actual, incrementamos el recuento de variables con cada iteración del ciclo para proporcionar un recuento continuo del número de argumentos procesados y, finalmente, ejecutamos un turno para cargar $1 con el siguiente argumento. Aquí está el programa en funcionamiento:

```
[me@linuxbox~]$ posit-param2 a b c d
Argument 1 = a
Argument 2 = b
Argument 3 = c
Argument 4 = d
```

Simple Applications
Aplicaciones sencillas

Even without shift , it's possible to write useful applications using positional parameters. By way of example, here is a simple file information program:
Incluso sin desplazamiento, es posible escribir aplicaciones útiles utilizando parámetros posicionales. A modo de ejemplo, aquí hay un programa de información de archivo simple:

```
#!/bin/bash

# file-info: simple file information program

PROGNAME="$(basename "$0")"

if [[ -e "$1" ]]; then
    echo -e "\nFile Type:"
    file "$1"
    echo -e "\nFile Status:"
    stat "$1"
else
    echo "$PROGNAME: usage: $PROGNAME file" >&2
    exit 1
fi
```

This program displays the file type (determined by the file command) and the file status (from the stat command) of a specified file. One interesting feature of this program is the PROGNAME variable. It is given the value that results from the basename "$0" command. The basename command removes the leading portion of a pathname, leaving only the base name of a file. In our example, basename removes the leading

portion of the pathname contained in the $0 parameter, the full pathname of our example program. This value is useful when constructing messages such as the usage message at the end of the program. By coding it this way, the script can be renamed, and the message automatically adjusts to contain the name of the program.
Este programa muestra el tipo de archivo (determinado por el comando de archivo) y el estado del archivo (desde el comando stat) de un archivo especificado. Una característica interesante de este programa es la variable PROGNAME. Se le da el valor que resulta del comando basename "$ 0". El comando basename elimina la parte inicial de un nombre de ruta, dejando solo el nombre base de un archivo. En nuestro ejemplo, basename elimina la parte inicial del nombre de ruta contenido en el parámetro $ 0, el nombre de ruta completo de nuestro programa de ejemplo. Este valor es útil al construir mensajes como el mensaje de uso al final del programa. Al codificarlo de esta manera, se puede cambiar el nombre del script y el mensaje se ajusta automáticamente para contener el nombre del programa.

Using Positional Parameters with Shell Functions
Usar parámetros posicionales con funciones de shell

Just as positional parameters are used to pass arguments to shell scripts, they can also be used to pass arguments to shell functions. To demonstrate, we will convert the file_info script into a shell function.
Así como los parámetros posicionales se utilizan para pasar argumentos a scripts de shell, también se pueden utilizar para pasar argumentos a funciones de shell. Para demostrarlo, convertiremos el script file_info en una función de shell.

```
file_info () {
    # file_info: function to display file information
    if [[ -e "$1" ]]; then
        echo -e "\nFile Type:"
        file "$1"
        echo -e "\nFile Status:"
        stat "$1"
    else
        echo "$FUNCNAME: usage: $FUNCNAME file" >&2
        return 1
    fi
}
```

Now, if a script that incorporates the file_info shell function calls the function with a filename argument, the argument will be passed to the function.
Ahora, si un script que incorpora la función de shell file_info llama a la función con un argumento de nombre de archivo, el argumento se pasará a la función.

With this capability, we can write many useful shell functions that not only can be used in scripts but also can be used within our .bashrc files. Notice that the PROGNAME variable was changed to the shell variable FUNCNAME . The shell automatically updates this variable to keep track of the currently executed shell

function. Note that $0 always contains the full pathname of the first item on the command line (i.e., the name of the program) and does not contain the name of the shell function as we might expect.
Con esta capacidad, podemos escribir muchas funciones de shell útiles que no solo se pueden usar en scripts sino que también se pueden usar dentro de nuestros archivos .bashrc. Observe que la variable PROGNAME se cambió a la variable de shell FUNCNAME. El shell actualiza automáticamente esta variable para realizar un seguimiento de la función del shell actualmente ejecutada. Tenga en cuenta que $ 0 siempre contiene la ruta completa del primer elemento en la línea de comandos (es decir, el nombre del programa) y no contiene el nombre de la función de shell como podríamos esperar.

Handling Positional Parameters en Masse
Manejo de parámetros posicionales en masa

It is sometimes useful to manage all the positional parameters as a group. For example, we might want to write a “wrapper” around another program.
A veces es útil administrar todos los parámetros posicionales como un grupo. Por ejemplo, podríamos querer escribir un "envoltorio" alrededor de otro programa.

This means we create a script or shell function that simplifies the invocation of another program. The wrapper, in this case, supplies a list of arcane command line options and then passes a list of arguments to the lower-level program.
Esto significa que creamos un script o función de shell que simplifica la invocación de otro programa. El contenedor, en este caso, proporciona una lista de opciones de línea de comandos arcanas y luego pasa una lista de argumentos al programa de nivel inferior.

The shell provides two special parameters for this purpose. They both expand into the complete list of positional parameters but differ in rather subtle ways. Table 32-1 describes these parameters.
El shell proporciona dos parámetros especiales para este propósito. Ambos se expanden en la lista completa de parámetros posicionales, pero difieren en formas bastante sutiles. La Tabla 32-1 describe estos parámetros.

> Table 32-1: The * and @ Special Parameters
> Tabla 32-1: Los parámetros especiales * y @

Parameter	Description
$*	Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands into a double-quoted string containing all of the positional parameters, each separated by the first character of the IFS shell variable (by default a space character). Se expande a la lista de parámetros posicionales, comenzando con 1. Cuando está rodeado por comillas dobles, se expande en una cadena entre comillas dobles que contiene todos los parámetros posicionales, cada uno separado por el primer carácter de la variable de shell IFS (por defecto, un carácter de espacio).

Parameter	Description
$@	Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands each positional parameter into a separate word as if it was surrounded by double quotes. Se expande a la lista de parámetros posicionales, comenzando con 1. Cuando está rodeado por comillas dobles, expande cada parámetro posicional en una palabra separada como si estuviera rodeado por comillas dobles.

Here is a script that shows these special parameters in action:
Aquí hay un script que muestra estos parámetros especiales en acción:

```
#!/bin/bash

# posit-params3: script to demonstrate $* and $@

print_params ()
    echo "\$1 = $1"
    echo "\$2 = $2"
    echo "\$3 = $3"
    echo "\$4 = $4"
}

{
pass_params () {
    echo -e "\n" '$* :'; print_params $*
    echo -e "\n" '"$*" :'; print_params "$*"
    echo -e "\n" '$@ :'; print_params $@
    echo -e "\n" '"$@" :'; print_params "$@"
}

pass_params "word" "words with spaces"
```

In this rather convoluted program, we create two arguments, called word and words with spaces , and pass them to the pass_params function. That function, in turn, passes them on to the print_params function, using each of the four methods available with the special parameters $* and $@ . When executed, the script reveals the differences.
En este programa bastante complicado, creamos dos argumentos, llamados palabra y palabras con espacios, y los pasamos a la función pass_params. Esa función, a su vez, los pasa a la función print_params, usando cada uno de los cuatro métodos disponibles con los parámetros especiales $ * y $ @. Cuando se ejecuta, el script revela las diferencias.

```
[me@linuxbox ~]$ posit-param3
$* :
$1 = word
```

```
$2 = words
$3 = with
$4 = spaces

"$*" :
$1 = word words with spaces
$2 =
$3 =
$4 =

$@ :
$1 = word
$2 = words
$3 = with
$4 = spaces

"$@" :
$1 = word
$2 = words with spaces
$3 =
$4 =
```

With our arguments, both $* and $@ produce a four-word result.
Con nuestros argumentos, tanto $* como $@ producen un resultado de cuatro palabras.

---

word words with spaces

"$*" produces a one-word result.

"word words with spaces"

"$@" produces a two-word result.

"word" "words with spaces"

This matches our actual intent. The lesson to take from this is that even though the shell provides four different ways of getting the list of positional parameters, "$@" is by far the most useful for most situations because it preserves the integrity of each positional parameter. To ensure safety, it should always be used, unless we have a compelling reason not to use it.
Esto coincide con nuestra intención real. La lección que se puede aprender de esto es que, aunque el shell proporciona cuatro formas diferentes de obtener la lista de parámetros posicionales, "$@" es, con mucho, el más útil para la mayoría de situaciones porque preserva la integridad de cada parámetro posicional. Para garantizar la seguridad, siempre debe usarse, a menos que tengamos una razón convincente para no usarlo.

A More Complete Application
Una aplicación más completa

After a long hiatus, we are going to resume work on our sys_info_page program, last seen in Chapter 27. Our next addition will add several command line options to the program as follows:
Después de una larga pausa, vamos a reanudar el trabajo en nuestro programa sys_info_page, visto por última vez en el Capítulo 27. Nuestra próxima adición agregará varias opciones de línea de comando al programa de la siguiente manera:

Output file We will add an option to specify a name for a file to contain the program's output. It will be specified as either -f file or --file file .
Archivo de salida Agregaremos una opción para especificar un nombre para un archivo que contenga la salida del programa. Se especificará como -f file o --file file.

Interactive mode This option will prompt the user for an output file-name and will determine whether the specified file already exists. If it does, the user will be prompted before the existing file is overwritten. This option will be specified by either -i or --interactive.
Modo interactivo Esta opción solicitará al usuario un nombre de archivo de salida y determinará si el archivo especificado ya existe. Si es así, se le preguntará al usuario antes de que se sobrescriba el archivo existente. Esta opción se especificará mediante -i o --interactive.

Help Either -h or --help may be specified to cause the program to output an informative usage message.Here is the code needed to implement the command line processing:
Ayuda Se puede especificar -ho --help para que el programa genere un mensaje de uso informativo. Aquí está el código necesario para implementar el procesamiento de la línea de comandos:

```
usage () {
    echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
    return
}

# process command line options

interactive=
filename=

while [[ -n "$1" ]]; do
    case "$1" in
        -f | --file)            shift
                                filename="$1"
                                ;;
        -i | --interactive)     interactive=1
                                ;;
        -h | --help)            usage
                                exit
                                ;;
        *)                      usage >&2
```

```
                                exit 1
                                ;;
    esac
    shift
done
```

First, we add a shell function called usage to display a message when the help option is invoked or an unknown option is attempted.
Primero, agregamos una función de shell llamada uso para mostrar un mensaje cuando se invoca la opción de ayuda o se intenta una opción desconocida.

Next, we begin the processing loop. This loop continues while the positional parameter $1 is not empty. At the end of the loop, we have a shift command to advance the positional parameters to ensure that the loop will eventually terminate.
A continuación, comenzamos el ciclo de procesamiento. Este bucle continúa mientras el parámetro posicional $ 1 no esté vacío. Al final del ciclo, tenemos un comando de cambio para avanzar los parámetros posicionales para asegurar que el ciclo eventualmente terminará.

Within the loop, we have a case statement that examines the current positional parameter to see whether it matches any of the supported choices. If a supported parameter is found, it is acted upon. If an unknown choice is found, the usage message is displayed, and the script terminates with an error.
Dentro del ciclo, tenemos una declaración de caso que examina el parámetro posicional actual para ver si coincide con alguna de las opciones admitidas. Si se encuentra un parámetro compatible, se actúa sobre él. Si se encuentra una opción desconocida, se muestra el mensaje de uso y el script termina con un error.

The -f parameter is handled in an interesting way. When detected, it causes an additional shift to occur, which advances the positional parameter $1 to the filename argument supplied to the -f option.
El parámetro -f se maneja de una manera interesante. Cuando se detecta, provoca un cambio adicional, que avanza el parámetro posicional $ 1 al argumento del nombre de archivo proporcionado a la opción -f.

We next add the code to implement the interactive mode.
A continuación, agregamos el código para implementar el modo interactivo.

```
# interactive mode

if [[ -n "$interactive" ]]; then
    while true; do
        read -p "Enter name of output file: " filename
        if [[ -e "$filename" ]]; then
            read -p "'$filename' exists. Overwrite? [y/n/q] > "
            case "$REPLY" in
                Y|y)    break
                        ;;
                Q|q)    echo "Program terminated."
                        exit
                        ;;
```

```
                *)      continue
                        ;;
            esac
        elif [[ -z "$filename" ]]; then
            continue
        else
            break
        fi
    done
fi
```

If the interactive variable is not empty, an endless loop is started, which contains the filename prompt and subsequent existing file-handling code. If the desired output file already exists, the user is prompted to overwrite, choose another filename, or quit the program. If the user chooses to overwrite an existing file, a break is executed to terminate the loop. Notice how the case statement detects only whether the user chooses to overwrite or quit. Any other choice causes the loop to continue and prompts the user again.
Si la variable interactiva no está vacía, se inicia un bucle sin fin, que contiene la solicitud de nombre de archivo y el código de manejo de archivos existente subsiguiente. Si el archivo de salida deseado ya existe, se le pide al usuario que lo sobrescriba, elija otro nombre de archivo o salga del programa. Si el usuario elige sobrescribir un archivo existente, se ejecuta una pausa para terminar el ciclo. Observe cómo la declaración del caso solo detecta si el usuario elige sobrescribir o salir. Cualquier otra opción hace que el bucle continúe y vuelve a solicitar al usuario.

To implement the output filename feature, we must first convert the existing page-writing code into a shell function, for reasons that will become clear in a moment.
Para implementar la función de nombre de archivo de salida, primero debemos convertir el código de escritura de página existente en una función de shell, por razones que se aclararán en un momento.

```
write_html_page () {
    cat <<- _EOF_
    <html>
        <head>
            <title>$TITLE</title>
        </head>
        <body>
            <h1>$TITLE</h1>
            <p>$TIMESTAMP</p>
            $(report_uptime)
            $(report_disk_space)
            $(report_home_space)
        </body>
    </html>
_EOF_
    return
}

# output html page
```

```
if [[ -n "$filename" ]]; then
    if touch "$filename" && [[ -f "$filename" ]]; then
        write_html_page > "$filename"
    else
        echo "$PROGNAME: Cannot write file '$filename'" >&2
        exit 1
    fi
else
    write_html_page
fi
```

The code that handles the logic of the -f option appears at the end of the previous listing. In it, we test for the existence of a filename, and if one is found, a test is performed to see whether the file is indeed writable. To do this, a touch is performed, followed by a test to determine whether the resulting file is a regular file. These two tests take care of situations where an invalid pathname is input (touch will fail), and, if the file already exists, that it's a regular file.
El código que maneja la lógica de la opción -f aparece al final de la lista anterior. En él, probamos la existencia de un nombre de archivo, y si se encuentra uno, se realiza una prueba para ver si el archivo es realmente modificable. Para hacer esto, se realiza un toque, seguido de una prueba para determinar si el archivo resultante es un archivo normal. Estas dos pruebas se encargan de situaciones en las que se ingresa un nombre de ruta no válido (el tacto fallará) y, si el archivo ya existe, que es un archivo normal.

As we can see, the write_html_page function is called to perform the actual generation of the page. Its output is either directed to standard output (if the variable filename is empty) or redirected to the specified file. Since we have two possible destinations for the HTML code, it makes sense to convert the write_html_page routine to a shell function to avoid redundant code.
Como podemos ver, se llama a la función write_html_page para realizar la generación real de la página. Su salida se dirige a la salida estándar (si el nombre de archivo de la variable está vacío) o se redirige al archivo especificado. Dado que tenemos dos posibles destinos para el código HTML, tiene sentido convertir la rutina write_html_page en una función de shell para evitar código redundante.

Summing Up
Resumen

With the addition of positional parameters, we can now write fairly functional scripts. For simple, repetitive tasks, positional parameters make it possible to write very useful shell functions that can be placed in a user's .bashrc file.
Con la adición de parámetros posicionales, ahora podemos escribir scripts bastante funcionales. Para tareas simples y repetitivas, los parámetros posicionales hacen posible escribir funciones de shell muy útiles que se pueden colocar en el archivo .bashrc de un usuario.

Our sys_info_page program has grown in complexity and sophistication.
Nuestro programa sys_info_page ha crecido en complejidad y sofisticación.

Here is a complete listing, with the most recent changes highlighted:
Aquí hay una lista completa, con los cambios más recientes resaltados:

```
#!/bin/bash

# sys_info_page: program to output a system information page

PROGNAME="$(basename "$0")"
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME="$(date +"%x %r %Z")"
TIMESTAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
    cat <<- _EOF_
        <h2>System Uptime</h2>
        <pre>$(uptime)</pre>
_EOF_
    return
}

report_disk_space () {
    cat <<- _EOF_
        <h2>Disk Space Utilization</h2>
        <pre>$(df -h)</PRE>
_EOF_
    return
}

report_home_space () {
    if [[ "$(id -u)" -eq 0 ]]; then
        cat <<- _EOF_
            <h2>Home Space Utilization (All Users)</h2>
            <pre>$(du -sh /home/*)</pre>
_EOF_
    else
        cat <<- _EOF_
            <h2>Home Space Utilization ($USER)</h2>
            <pre>$(du -sh "$HOME")</pre>
_EOF_
    fi
    return
}

usage () {
    echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
    return
}

write_html_page () {
    cat <<- _EOF_
        <html>
            <head>
                <title>$TITLE</title>
            </head>
```

```
            <body>
                <h1>$TITLE</h1>
                <p>$TIMESTAMP</p>
                $(report_uptime)
                $(report_disk_space)
                $(report_home_space)
            </body>
        </html>
_EOF_
    return
}

# process command line options

interactive=
filename=

while [[ -n "$1" ]]; do
    case "$1" in
        -f | --file)    shift
                        filename="$1"
                        ;;
        -i | --interactive) interactive=1
                            ;;
        -h | --help)    usage
                        exit
                        ;;
        *)  usage >&2
            exit 1
            ;;
    esac
    shift
done

# interactive mode

if [[ -n "$interactive" ]]; then
    while true; do
        read -p "Enter name of output file: " filename
        if [[ -e "$filename" ]]; then
            read -p "'$filename' exists. Overwrite? [y/n/q] > "
            case "$REPLY" in
                Y|y)    break
                        ;;
                Q|q)    echo "Program terminated."
                        exit
                        ;;
                *)  continue
                    ;;
            esac
        elif [[ -z "$filename" ]]; then
            continue
        else
            break
```

```
        fi
    done
fi

# output html page
if [[ -n "$filename" ]]; then
    if touch "$filename" && [[ -f "$filename" ]]; then
        write_html_page > "$filename"
    else
        echo "$PROGNAME: Cannot write file '$filename'" >&2
        exit 1
    fi
else
    write_html_page
fi
```

We're not done yet. There are still a few more things we can do and improvements we can make.
Aún no hemos terminado. Todavía hay algunas cosas más que podemos hacer y mejoras que podemos realizar.