# 27

# Flow Control: Branching with if
# Control de flujo: ramificación con if

In the previous chapter, we were presented with a problem. How can we make our report-generator script adapt to the privileges of the user running the script? The solution to this problem will require us to find a way to “change directions” within our script, based on the results of a test. In programming terms, we need the program to branch.
En el capítulo anterior, se nos presentó un problema. ¿Cómo podemos hacer que nuestro script generador de informes se adapte a los privilegios del usuario que ejecuta el script? La solución a este problema requerirá que encontremos una manera de "cambiar de dirección" dentro de nuestro script, en función de los resultados de una prueba. En términos de programación, necesitamos que el programa se ramifique.

Let’s consider a simple example of logic expressed in pseudocode, a simulation of a computer language intended for human consumption.
Consideremos un ejemplo simple de lógica expresada en pseudocódigo, una simulación de un lenguaje de computadora destinado al consumo humano.

```
X = 5
If X = 5, then:
    Say “X equals 5.”
Otherwise:
    Say “X is not equal to 5.”
```

This is an example of a branch. Based on the condition “Does X = 5?” do one thing, “Say X equals 5,” and otherwise do another thing, “Say X is not equal to 5.”
Este es un ejemplo de una rama. Basado en la condición "¿X = 5?" hacer una cosa, "Diga que X es igual a 5" y, de lo contrario, haga otra cosa, "Diga que X no es igual a 5".

## if Statements
## declaraciones if

Using the shell, we can code the previous logic as follows:
Usando el shell, podemos codificar la lógica anterior de la siguiente manera:

```
x=5
if [ "$x" -eq 5 ]; then
    echo "x equals 5."
else
    echo "x does not equal 5."
fi
```

---

Or we can enter it directly at the command line (slightly shortened).
O podemos ingresarlo directamente en la línea de comando (ligeramente abreviado).

---

```
[~]$ x=5
[~]$ if [ "$x" -eq 5 ]; then echo "equals 5"; else echo "does not equal 5";
fi
equals 5
[~]$ x=0
[~]$ if [ "$x" -eq 5 ]; then echo "equals 5"; else echo "does not equal 5";
fi
does not equal 5
```

---

In this example, we execute the command twice: once, with the value of x set to 5, which results in the string “equals 5” being output, and the second time with the value of x set to 0, which results in the string “does not equal 5” being output.
En este ejemplo, ejecutamos el comando dos veces: una vez, con el valor de x establecido en 5, lo que da como resultado la salida de la cadena "igual a 5", y la segunda vez con el valor de x establecido en 0, lo que da como resultado el la cadena "no es igual a 5" se emite.

The if statement has the following syntax:
La instrucción if tiene la siguiente sintaxis:

---

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi
```

---

where commands is a list of commands. This is a little confusing at first glance.
donde comandos es una lista de comandos. Esto es un poco confuso a primera vista.

But before we can clear this up, we have to look at how the shell evaluates the success or failure of a command.
Pero antes de que podamos aclarar esto, tenemos que ver cómo evalúa el shell el éxito o el fracaso de un comando.

# Exit Status
# Estado de salida

Commands (including the scripts and shell functions we write) issue a value to the system when they terminate, called an exit status. This value, which is an integer in the range of 0 to 255, indicates the success or failure of the command's execution. By convention, a value of zero indicates success and any other value indicates failure. The shell provides a parameter that we can use to examine the exit status. Here we see it in action:
Los comandos (incluidos los scripts y las funciones de shell que escribimos) emiten un valor para el sistema cuando terminan, llamado estado de salida. Este valor, que es un número entero en el rango de 0 a 255, indica el éxito o el fracaso de la ejecución del comando. Por convención, un valor de cero indica éxito y cualquier otro valor indica fracaso. El shell proporciona un parámetro que podemos usar para examinar el estado de salida. Aquí lo vemos en acción:

```
[~]$ ls -d /usr/bin
/usr/bin
[~]$ echo $?
0
[~]$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
[~]$ echo $?
2
```

In this example, we execute the ls command twice. The first time, the command executes successfully. If we display the value of the parameter $?, we see that it is zero. We execute the ls command a second time (specifying a nonexistent directory), producing an error, and examine the parameter $? again. This time it contains a 2, indicating that the command encountered an error. Some commands use different exit status values to provide diagnostics for errors, while many commands simply exit with a value of 1 when they fail. Man pages often include a section entitled "Exit Status," describing what codes are used. However, a zero always indicates success.
En este ejemplo, ejecutamos el comando ls dos veces. La primera vez, el comando se ejecuta correctamente. Si mostramos el valor del parámetro $ ?, vemos que es cero. Ejecutamos el comando ls por segunda vez (especificando un directorio inexistente), produciendo un error, y examinamos el parámetro $? de nuevo. Esta vez contiene un 2, lo que indica que el comando encontró un error. Algunos comandos utilizan diferentes valores de estado de salida para proporcionar diagnósticos de errores, mientras que muchos comandos simplemente salen con un valor de 1 cuando fallan. Las páginas de manual a menudo incluyen una sección titulada "Estado de salida", que describe qué códigos se utilizan. Sin embargo, un cero siempre indica éxito.

The shell provides two extremely simple builtin commands that do nothing except terminate with either a 0 or 1 exit status. The true command always executes successfully, and the false command always executes unsuccessfully.
El shell proporciona dos comandos integrados extremadamente simples que no hacen nada más que terminar con un estado de salida 0 o 1. El comando verdadero siempre se ejecuta con éxito y el comando falso siempre se ejecuta sin éxito.

---

```
[~]$ true
[~]$ echo $?
0
[~]$ false
[~]$ echo $?
1
```

---

We can use these commands to see how the if statement works. What the if statement really does is evaluate the success or failure of commands.
Podemos usar estos comandos para ver cómo funciona la instrucción if. Lo que realmente hace la declaración if es evaluar el éxito o el fracaso de los comandos.

---

```
[~]$ if true; then echo "It's true."; fi
It's true.
[~]$ if false; then echo "It's true."; fi
[~]$
```

---

The command echo "It's true." is executed when the command following if executes successfully and is not executed when the command following if does not execute successfully. If a list of commands follows if , the last command in the list is evaluated.
El comando echo "Es verdad". se ejecuta cuando el comando siguiente if se ejecuta correctamente y no se ejecuta cuando el comando siguiente if no se ejecuta correctamente. Si sigue una lista de comandos, se evalúa el último comando de la lista.

---

```
[~]$ if false; true; then echo "It's true."; fi
It's true.
[~]$ if true; false; then echo "It's true."; fi
[~]$
```

---

# Using test
# Usando prueba

By far, the command used most frequently with if is test . The test command performs a variety of checks and comparisons. It has two equivalent forms. The first, shown here:
De lejos, el comando que se usa con más frecuencia con if es test. El comando de prueba realiza una variedad de comprobaciones y comparaciones. Tiene dos formas equivalentes. El primero, mostrado aquí:

```
test expression
```

And the second, more popular form, here:
Y la segunda forma, más popular, aquí:

```
[ expression ]
```

where expression is an expression that is evaluated as either true or false.
donde expresión es una expresión que se evalúa como verdadera o falsa.

The test command returns an exit status of 0 when the expression is true and a status of 1 when the expression is false.
El comando de prueba devuelve un estado de salida de 0 cuando la expresión es verdadera y un estado de 1 cuando la expresión es falsa.

It is interesting to note that both test and [ are actually commands. In bash they are builtins, but they also exist as programs in /usr/bin for use with other shells. The expression is actually just its arguments with the [ command requiring that the ] character be provided as its final argument.
Es interesante notar que tanto test como [son en realidad comandos. En bash son incorporados, pero también existen como programas en / usr / bin para usar con otros shells. La expresión es en realidad solo sus argumentos con el [comando que requiere que se proporcione el carácter] como su argumento final.

The test and [ commands support a wide range of useful expressions and tests.
Los comandos test y [admiten una amplia gama de expresiones y pruebas útiles.

## File Expressions
## Expresiones de archivo

Table 27-1 lists the expressions used to evaluate the status of files.
La Tabla 27-1 enumera las expresiones utilizadas para evaluar el estado de los archivos.

Table 27-1: test File Expressions
Tabla 27-1: Expresiones de archivo de prueba

| Expression | Is true if: |
|---|---|

| Expression | Is true if: |
|---|---|
| file1 -ef file2 | file1 and file2 have the same inode numbers (the two filenames refer to the same file by hard linking).<br>file1 y file2 tienen los mismos números de inodo (los dos nombres de archivo se refieren al mismo archivo mediante enlaces físicos). |
| file1 -nt file2 | file1 is newer than file2.<br>file1 es más reciente que file2. |
| file1 -ot file2 | file1 is older than file2.<br>file1 es más antiguo que file2. |
| -b file | file exists and is a block-special (device) file.<br>El archivo existe y es un archivo de bloque especial (dispositivo). |
| -c file | file exists and is a character-special (device) file.<br>El archivo existe y es un archivo de carácter especial (dispositivo). |
| -d file | file exists and is a directory.<br>El archivo existe y es un directorio. |
| -e file | file exists.<br>El archivo existe |
| -f file | file exists and is a regular file.<br>El archivo existe y es un archivo normal. |
| -g file | file exists and is set-group-ID.<br>El archivo existe y es set-group-ID. |
| -G file | file exists and is owned by the effective group ID.<br>El archivo existe y es propiedad del ID de grupo efectivo. |
| -k file | file exists and has its “sticky bit” set.<br>archivo existe y tiene su "bit pegajoso" establecido. |
| -L file | file exists and is a symbolic link.<br>El archivo existe y es un enlace simbólico. |
| -O file | file exists and is owned by the effective user ID.<br>El archivo existe y es propiedad del ID de usuario efectivo. |
| -p file | file exists and is a named pipe.<br>El archivo existe y es una tubería con nombre. |
| -r file | file exists and is readable (has readable permission for the effective user).<br>El archivo existe y es legible (tiene permiso de lectura para el usuario efectivo). |
| -s file | file exists and has a length greater than zero.<br>El archivo existe y tiene una longitud mayor que cero. |
| -S file | file exists and is a network socket.<br>El archivo existe y es un enchufe de red. |

| Expression | Is true if: |
|---|---|
| -t fd | fd is a file descriptor directed to/from the terminal. This can be used to determine whether standard input/output/error is being redirected.<br>fd es un descriptor de archivo dirigido hacia / desde la terminal. Esto se puede utilizar para determinar si se está redirigiendo la entrada / salida / error estándar. |
| -u file | file exists and is setuid.<br>el archivo existe y es setuid. |
| -w file | file exists and is writable (has write permission for the effective user).<br>El archivo existe y se puede escribir (tiene permiso de escritura para el usuario efectivo). |
| -x file | file exists and is executable (has execute/search permission for the effective user).<br>El archivo existe y es ejecutable (tiene permiso de ejecución / búsqueda para el usuario efectivo). |

Here we have a script that demonstrates some of the file expressions:
Aquí tenemos un script que demuestra algunas de las expresiones del archivo:

```
#!/bin/bash

# test-file: Evaluate the status of a file

FILE=~/.bashrc

if [ -e "$FILE" ]; then
    if [ -f "$FILE" ]; then
        echo "$FILE is a regular file."
    fi
    if [ -d "$FILE" ]; then
        echo "$FILE is a directory."
    fi
    if [ -r "$FILE" ]; then
        echo "$FILE is readable."
    fi
    if [ -w "$FILE" ]; then
        echo "$FILE is writable."
    fi
    if [ -x "$FILE" ]; then
        echo "$FILE is executable/searchable."
    fi
else
    echo "$FILE does not exist"
    exit 1
fi

exit
```

The script evaluates the file assigned to the constant FILE and displays its results as the evaluation is performed. There are two interesting things to note about this script. First, notice how the parameter $FILE is quoted within the expressions. This is not required to syntactically complete the expression; rather, it is a defense against the parameter being empty or containing only whitespace. If the parameter expansion of $FILE were to result in an empty value, it would cause an error (the operators would be interpreted as non-null strings rather than operators). Using the quotes around the parameter ensures that the operator is always followed by a string, even if the string is empty. Second, notice the presence of the exit command near the end of the script. The exit command accepts a single, optional argument, which becomes the script's exit status. When no argument is passed, the exit status defaults to the exit status of the last command executed. Using exit in this way allows the script to indicate failure if $FILE expands to the name of a nonexistent file. The exit command appearing on the last line of the script is there as a formality. When a script "runs off the end" (reaches end of file), it terminates with an exit status of the last command executed.
El script evalúa el archivo asignado a la constante FILE y muestra sus resultados a medida que se realiza la evaluación. Hay dos cosas interesantes a tener en cuenta sobre este guión. Primero, observe cómo se cita el parámetro $ FILE dentro de las expresiones. Esto no es necesario para completar sintácticamente la expresión; más bien, es una defensa contra que el parámetro esté vacío o contenga solo espacios en blanco. Si la expansión del parámetro de $ FILE resultara en un valor vacío, causaría un error (los operadores se interpretarían como cadenas no nulas en lugar de operadores). El uso de comillas alrededor del parámetro asegura que el operador siempre esté seguido por una cadena, incluso si la cadena está vacía. En segundo lugar, observe la presencia del comando de salida cerca del final del script. El comando de salida acepta un único argumento opcional, que se convierte en el estado de salida del script. Cuando no se pasa ningún argumento, el estado de salida cambia por defecto al estado de salida del último comando ejecutado. El uso de exit de esta manera permite que el script indique un error si $ FILE se expande al nombre de un archivo inexistente. El comando de salida que aparece en la última línea del script está ahí como formalidad. Cuando un script "se ejecuta al final" (llega al final del archivo), termina con un estado de salida del último comando ejecutado.

Similarly, shell functions can return an exit status by including an integer argument to the return command. If we were to convert the previous script to a shell function to include it in a larger program, we could replace the exit commands with return statements and get the desired behavior.
De manera similar, las funciones de shell pueden devolver un estado de salida al incluir un argumento entero en el comando de retorno. Si convirtiéramos el script anterior en una función de shell para incluirlo en un programa más grande, podríamos reemplazar los comandos de salida con declaraciones de retorno y obtener el comportamiento deseado.

```
test_file () {
    # test-file: Evaluate the status of a file

    FILE=~/.bashrc

    if [ -e "$FILE" ]; then
        if [ -f "$FILE" ]; then
            echo "$FILE is a regular file."
        fi
```

```
        if [ -d "$FILE" ]; then
            echo "$FILE is a directory."
        fi
        if [ -r "$FILE" ]; then
            echo "$FILE is readable."
        fi
        if [ -w "$FILE" ]; then
            echo "$FILE is writable."
        fi
        if [ -x "$FILE" ]; then
            echo "$FILE is executable/searchable."
        fi
    else
        echo "$FILE does not exist"
        return 1
    fi

}
```

# String Expressions
## Expresiones de cadena

Table 27-2 lists the expressions used to evaluate strings.
La Tabla 27-2 enumera las expresiones utilizadas para evaluar cadenas.

> Table 27-2: test String Expressions
> Tabla 27-2: Expresiones de cadena de prueba

| Expression | Is true if: |
|---|---|
| string | string is not null.<br>la cadena no es nula. |
| -n string | The length of string is greater than zero.<br>La longitud de la cuerda es mayor que cero. |
| -z string | The length of string is zero.<br>La longitud de la cuerda es cero. |
| string1 = string2<br>string1 == string2 | string1 and string2 are equal. Single or double equal signs may be used. The use of double equal signs is greatly preferred, but it is not POSIX compliant.<br>string1 y string2 son iguales. Se pueden usar signos iguales simples o dobles. Se prefiere mucho el uso de signos dobles iguales, pero no es compatible con POSIX. |
| string1 != string2 | string1 and string2 are not equal.<br>string1 y string2 no son iguales. |

| Expression | Is true if: |
|---|---|
| string1 > string2 | string1 sorts after string2.<br>string1 ordena después de string2. |
| string1 < string2 | string1 sorts before string2.<br>string1 se ordena antes que string2. |

**Warning**
**Advertencia**

***The > and < expression operators must be quoted (or escaped with a backslash) when used with test . If they are not, they will be interpreted by the shell as redirection operators, with potentially destructive results. Also note that while the bash documentation states that the sorting order conforms to the collation order of the current locale, it does not. ASCII (POSIX) order is used in versions of bash up to and including 4.0. This problem was fixed in version 4.1.***
***Los operadores de expresión > y < deben estar entrecomillados (o escapados con una barra invertida) cuando se usan con test. Si no es así, el shell los interpretará como operadores de redirección, con resultados potencialmente destructivos. También tenga en cuenta que si bien la documentación de bash indica que el orden de clasificación se ajusta al orden de clasificación de la configuración regional actual, no es así. El orden ASCII (POSIX) se usa en versiones de bash hasta 4.0 inclusive. Este problema se solucionó en la versión 4.1.***

Here is a script that incorporates string expressions:
Aquí hay un script que incorpora expresiones de cadena:

```
#!/bin/bash

# test-string: evaluate the value of a string

ANSWER=maybe

if [ -z "$ANSWER" ]; then
    echo "There is no answer." >&2
    exit 1
fi

if [ "$ANSWER" = "yes" ]; then
    echo "The answer is YES."
elif [ "$ANSWER" = "no" ]; then
    echo "The answer is NO."
elif [ "$ANSWER" = "maybe" ]; then
    echo "The answer is MAYBE."
else
    echo "The answer is UNKNOWN."
fi
```

In this script, we evaluate the constant ANSWER . We first determine whether the string is empty. If it is, we terminate the script and set the exit status to 1.
En este script, evaluamos la constante RESPUESTA. Primero determinamos si la cadena está vacía. Si es así, finalizamos el script y establecemos el estado de salida en 1.

Notice the redirection that is applied to the echo command. This redirects the error message “There is no answer.” to standard error, which is the proper thing to do with error messages. If the string is not empty, we evaluate the value of the string to see whether it is equal to either “yes,” “no,” or “maybe.” We do this by using elif , which is short for “else if.” By using elif , we are able to construct a more complex logical test.
Observe la redirección que se aplica al comando echo. Esto redirige el mensaje de error "No hay respuesta". al error estándar, que es lo correcto con los mensajes de error. Si la cadena no está vacía, evaluamos el valor de la cadena para ver si es igual a "sí", "no" o "tal vez". Hacemos esto usando elif, que es la abreviatura de "else if". Al usar elif, podemos construir una prueba lógica más compleja.

# Integer Expressions
# Expresiones enteras

To compare values as integers rather than as strings, we can use the expressions listed in Table 27-3.
Para comparar valores como números enteros en lugar de cadenas, podemos usar las expresiones enumeradas en la tabla 27-3.

> Table 27-3: test Integer Expressions
> Tabla 27-3: prueba de expresiones enteras

| Expression | Is true if: |
|---|---|
| integer1 -eq integer2 | integer1 is equal to integer2.<br>integer1 es igual a integer2. |
| integer1 -ne integer2 | integer1 is not equal to integer2.<br>integer1 no es igual a integer2. |
| integer1 -le integer2 | integer1 is less than or equal to integer2.<br>integer1 es menor o igual que integer2. |
| integer1 -lt integer2 | integer1 is less than integer2.<br>integer1 es menor que integer2. |
| integer1 -ge integer2 | integer1 is greater than or equal to integer2.<br>integer1 es mayor o igual que integer2. |
| integer1 -gt integer2 | integer1 is greater than integer2.<br>integer1 es mayor que integer2. |

Here is a script that demonstrates them:
Aquí hay un guión que los demuestra:

```
#!/bin/bash

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

INT=-5

if [ -z "$INT" ]; then
    echo "INT is empty." >&2
    exit 1
fi

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
```

---

The interesting part of the script is how it determines whether an integer is even or odd. By performing a modulo 2 operation on the number, which divides the number by 2 and returns the remainder, it can tell whether the number is odd or even.
La parte interesante del script es cómo determina si un número entero es par o impar. Al realizar una operación de módulo 2 en el número, que divide el número por 2 y devuelve el resto, puede saber si el número es par o impar.

# A More Modern Version of test
# Una versión más moderna de la prueba

---

Modern versions of bash include a compound command that acts as an enhanced replacement for test . It uses the following syntax:
Las versiones modernas de bash incluyen un comando compuesto que actúa como un reemplazo mejorado para la prueba. Utiliza la siguiente sintaxis:

---

```
[[ expression ]]
```

---

where, like test , expression is an expression that evaluates to either a true or false result. The [[ ]] command is similar to test (it supports all of its expressions) but adds an important new string expression.
donde, como prueba, expresión es una expresión que se evalúa como un resultado verdadero o falso. El comando [[]] es similar a test (admite todas sus expresiones) pero agrega una nueva expresión de cadena importante.

---

```
string1 =~ regex
```

---

This returns true if string1 is matched by the extended regular expression regex . This opens up a lot of possibilities for performing such tasks as data validation. In our earlier example of the integer expressions, the script would fail if the constant INT contained anything except an integer. The script needs a way to verify that the constant contains an integer.
Esto devuelve verdadero si string1 coincide con la expresión regular extendida regex. Esto abre muchas posibilidades para realizar tareas como la validación de datos. En nuestro ejemplo anterior de las expresiones enteras, el script fallaría si la constante INT contuviera algo excepto un entero. El script necesita una forma de verificar que la constante contiene un número entero.

Using [[ ]] with the =~ string expression operator, we could improve the script this way:
Usando [[ ]] con el operador de expresión de cadena = ~, podríamos mejorar el script de esta manera:

---

```
#!/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
```

By applying the regular expression, we are able to limit the value of INT to only strings that begin with an optional minus sign, followed by one or more numerals. This expression also eliminates the possibility of empty values.
Al aplicar la expresión regular, podemos limitar el valor de INT solo a las cadenas que comienzan con un signo menos opcional, seguido de uno o más números. Esta expresión también elimina la posibilidad de valores vacíos.

Another added feature of [[ ]] is that the == operator supports pattern matching the same way pathname expansion does. Here's an example:
Otra característica añadida de [[]] es que el operador == admite la coincidencia de patrones de la misma manera que lo hace la expansión de nombre de ruta. Aquí tienes un ejemplo:

```
[~]$ FILE=foo.bar
[~]$ if [[ $FILE == foo.* ]]; then
> echo "$FILE matches pattern 'foo.*'"
> fi
foo.bar matches pattern 'foo.*'
```

This makes [[ ]] useful for evaluating file and pathnames
Esto hace que [[ ]] sea útil para evaluar archivos y rutas

# (( )) - Designed for Integers
# (( ))- Diseñado para números enteros

In addition to the [[ ]] compound command, bash also provides the (( )) compound command, which is useful for operating on integers. It supports a full set of arithmetic evaluations, a subject we will cover fully in Chapter 34.
Además del comando compuesto [[]], bash también proporciona el comando compuesto (()), que es útil para operar con números enteros. Admite un conjunto completo de evaluaciones aritméticas, un tema que cubriremos por completo en el Capítulo 34.

(( )) is used to perform arithmetic truth tests. An arithmetic truth test results in true if the result of the arithmetic evaluation is non-zero.
(()) se utiliza para realizar pruebas aritméticas de verdad. Una prueba de verdad aritmética da como resultado verdadero si el resultado de la evaluación aritmética es distinto de cero.

```
[~]$ if ((1)); then echo "It is true."; fi
It is true.
[~]$ if ((0)); then echo "It is true."; fi
[~]$
```

Using (( )) , we can slightly simplify the test-integer2 script like this:
Usando (()), podemos simplificar ligeramente el script test-integer2 de esta manera:

```
#!/bin/bash

# test-integer2a: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if ((INT == 0)); then
        echo "INT is zero."
    else
        if ((INT < 0)); then
            echo "INT is negative."
        else
            echo "INT is positive."
        fi

        if (( ((INT % 2)) == 0)); then
            echo "INT is even."
        else
            echo "INT is odd."
        fi
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

Notice that we use less-than and greater-than signs and that == is used to test for equivalence. This is a more natural-looking syntax for working with integers. Notice too, that because the compound command (( )) is part of the shell syntax rather than an ordinary command and it deals only with integers, it is able to recognize variables by name and does not require expansion to be performed. We'll discuss (( )) and the related arithmetic expansion further in Chapter 34.
Observe que usamos los signos menor que y mayor que y que == se usa para probar la equivalencia. Esta es una sintaxis de apariencia más natural para trabajar con números enteros. Tenga en cuenta también que debido a que el comando compuesto (()) es parte de la sintaxis del shell en lugar de un comando ordinario y se ocupa solo de números enteros, puede reconocer las variables por nombre y no requiere que se realice una expansión. Discutiremos (()) y la expansión aritmética relacionada con más detalle en el Capítulo 34.

# Combining Expressions
# Combinando Expresiones

It's also possible to combine expressions to create more complex evaluations. Expressions are combined by using logical operators. We saw these in Chapter 17 when we learned about the find command. There are three logical operations for test and [[ ]] . They are AND, OR, and NOT. test and [[ ]] use different operators to represent these operations, as shown in Table 27-4.
También es posible combinar expresiones para crear evaluaciones más complejas. Las expresiones se combinan mediante operadores lógicos. Los vimos en el Capítulo 17 cuando aprendimos sobre el comando de búsqueda. Hay tres operaciones lógicas para prueba y [[]]. Son Y, O y NO. test y [[]] utilizan diferentes operadores para representar estas operaciones, como se muestra en la Tabla 27-4.

> Table 27-4: Logical Operators
> Tabla 27-4: Operadores lógicos

| Operation | test | [[ ]] and (( )) |
|---|---|---|
| AND | -a | && |
| OR | -o | \|\| |
| NOT | ! | ! |

Here's an example of an AND operation. The following script determines whether an integer is within a range of values:
A continuación, se muestra un ejemplo de una operación AND. La siguiente secuencia de comandos determina si un número entero está dentro de un rango de valores:

```
#!/bin/bash

# test-integer3: determine if an integer is within a

# specified range of values.

MIN_VAL=1
MAX_VAL=100
INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [[ "$INT" -ge "$MIN_VAL" && "$INT" -le "$MAX_VAL" ]]; then
        echo "$INT is within $MIN_VAL to $MAX_VAL."
    else
        echo "$INT is out of range."
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

In this script, we determine whether the value of integer INT lies between the values of MIN_VAL and MAX_VAL . This is performed by a single use of [[ ]], which includes two expressions separated by the &&

operator. We also could have coded this using test:
En este script, determinamos si el valor del entero INT se encuentra entre los valores de MIN_VAL y MAX_VAL. Esto se realiza mediante un solo uso de [[]], que incluye dos expresiones separadas por el operador &&. También podríamos haber codificado esto usando test:

---

```
if [ "$INT" -ge "$MIN_VAL" -a "$INT" -le "$MAX_VAL" ]; then
echo "$INT is within $MIN_VAL to $MAX_VAL."
else
echo "$INT is out of range."
fi
```

---

The ! negation operator reverses the outcome of an expression. It returns true if an expression is false, and it returns false if an expression is true. In the following script, we modify the logic of our evaluation to find values of INT that are outside the specified range:
Los ! El operador de negación invierte el resultado de una expresión. Devuelve verdadero si una expresión es falsa y devuelve falso si una expresión es verdadera. En el siguiente script, modificamos la lógica de nuestra evaluación para encontrar valores de INT que están fuera del rango especificado:

---

```
#!/bin/bash

# test-integer4: determine if an integer is outside a

# specified range of values.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [[ ! ("$INT" -ge "$MIN_VAL" && "$INT" -le "$MAX_VAL") ]]; then
        echo "$INT is outside $MIN_VAL to $MAX_VAL."
    else
        echo "$INT is in range."
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

---

We also include parentheses around the expression, for grouping. If these were not included, the negation would only apply to the first expression and not the combination of the two. Coding this with test would be done this way:
También incluimos paréntesis alrededor de la expresión, para agrupar. Si no se incluyen, la negación solo se

aplicaría a la primera expresión y no a la combinación de las dos. Codificar esto con prueba se haría de esta manera:

```
if [ ! \( "$INT" -ge "$MIN_VAL" -a "$INT" -le "$MAX_VAL" \) ]; then
    echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
    echo "$INT is in range."
fi
```

Since all expressions and operators used by test are treated as command arguments by the shell (unlike [[ ]] and (( )) ), characters that have special meaning to bash , such as <, >, (, and ), must be quoted or escaped.
Dado que todas las expresiones y operadores usados por la prueba son tratados como argumentos de comando por el shell (a diferencia de [[]] y ((())), los caracteres que tienen un significado especial para bash, como <,>, (y), deben ser citado o escapado.

Seeing that test and [[ ]] do roughly the same thing, which is preferable? test is traditional (and part of the POSIX specification for standard shells, which are often used for system startup scripts), whereas [[ ]] is specific to bash (and a few other modern shells). It's important to know how to use test since it is widely used, but [[ ]] is clearly more useful and is easier to code, so it is preferred for modern scripts.
Ver esa prueba y [[]] hacen aproximadamente lo mismo, ¿cuál es preferible? test es tradicional (y parte de la especificación POSIX para shells estándar, que a menudo se usan para scripts de inicio del sistema), mientras que [[]] es específico de bash (y algunos otros shells modernos). Es importante saber cómo usar la prueba, ya que se usa ampliamente, pero [[]] es claramente más útil y más fácil de codificar, por lo que se prefiere para los scripts modernos.

## Portability Is the Hobgoblin of Little Minds
## La portabilidad es el duende de las pequeñas mentes

> If you talk to "real" Unix people, you quickly discover that many of them don't like Linux very much. They regard it as impure and unclean. One tenet of Unix users is that everything should be "portable." This means that any script you write should be able to run, unchanged, on any Unix-like system.
> Si habla con gente "real" de Unix, rápidamente descubre que a muchos de ellos no les gusta mucho Linux. Lo consideran impuro e inmundo. Un principio de los usuarios de Unix es que todo debe ser "portátil". Esto significa que cualquier script que escriba debería poder ejecutarse, sin cambios, en cualquier sistema similar a Unix.

> Unix people have good reason to believe this. Having seen what proprietary extensions to commands and shells did to the Unix world before POSIX, they are naturally wary of the effect of Linux on their beloved OS.
> La gente de Unix tiene buenas razones para creer esto. Habiendo visto lo que las extensiones

> propietarias de comandos y shells hicieron en el mundo Unix antes de POSIX, naturalmente desconfían del efecto de Linux en su amado sistema operativo.

> But portability has a serious downside. It prevents progress. It requires that things are always done using “lowest common denominator” techniques. In the case of shell programming, it means making everything compatible with sh , the original Bourne shell.
> Pero la portabilidad tiene una seria desventaja. Impide el progreso. Requiere que las cosas se hagan siempre utilizando técnicas de "mínimo común denominador". En el caso de la programación de shell, significa hacer todo compatible con sh, el shell Bourne original.

> This downside is the excuse that proprietary software vendors use to justify their proprietary extensions, only they call them “innovations.” But they are really just lock-in devices for their customers.
> Esta desventaja es la excusa que usan los proveedores de software propietario para justificar sus extensiones patentadas, solo que ellos las llaman "innovaciones". Pero en realidad son solo dispositivos de bloqueo para sus clientes.

> The GNU tools, such as bash , have no such restrictions. They encourage portability by supporting standards and by being universally available. You can install bash and the other GNU tools on almost any kind of system, even Windows, without cost. So feel free to use all the features of bash . It’s really portable.
> Las herramientas GNU, como bash, no tienen tales restricciones. Fomentan la portabilidad al respaldar los estándares y al estar disponibles universalmente. Puede instalar bash y las otras herramientas GNU en casi cualquier tipo de sistema, incluso Windows, sin costo alguno. Así que siéntete libre de usar todas las funciones de bash. Es realmente portátil.

# Control Operators: Another Way to Branch
# Operadores de control: otra forma de bifurcar

bash provides two control operators that can perform branching. The && (AND) and || (OR) operators work like the logical operators in the [[ ]] compound command. Here is the syntax for && :
bash proporciona dos operadores de control que pueden realizar ramificaciones. && (AND) y || Los operadores (OR) funcionan como los operadores lógicos en el comando compuesto [[]]. Aquí está la sintaxis de &&:

```
command1 && command2
```

Here is the syntax for || :
Aquí está la sintaxis de || :

```
command1 || command2
```

It is important to understand the behavior of these. With the && operator, command1 is executed, and command2 is executed if, and only if, command1 is successful. With the || operator, command1 is executed and command2 is executed if, and only if, command1 is unsuccessful.
Es importante comprender el comportamiento de estos. Con el operador &&, el comando1 se ejecuta y el comando2 se ejecuta si, y solo si, el comando1 tiene éxito. Con el || operador, comando1 se ejecuta y comando2 se ejecuta si, y solo si, comando1 no tiene éxito.

In practical terms, it means that we can do something like this:
En términos prácticos, significa que podemos hacer algo como esto:

```
[me@linuxbox ~]$ mkdir temp && cd temp
```

This will create a directory named temp, and if it succeeds, the current working directory will be changed to temp. The second command is attempted only if the mkdir command is successful. Likewise, a command like this:
Esto creará un directorio llamado temp, y si tiene éxito, el directorio de trabajo actual se cambiará a temp. El segundo comando se intenta solo si el comando mkdir es exitoso. Asimismo, un comando como este:

```
[~]$ [[ -d temp ]] || mkdir temp
```

will test for the existence of the directory temp, and only if the test fails will the directory be created. This type of construct is handy for handling errors in scripts, a subject we will discuss more in later chapters. For example, we could do this in a script:
probará la existencia del directorio temporal, y solo si la prueba falla, se creará el directorio. Este tipo de construcción es útil para manejar errores en scripts, un tema que discutiremos más en capítulos posteriores. Por ejemplo, podríamos hacer esto en un script:

```
[ -d temp ] || exit 1
```

If the script requires the directory temp and it does not exist, then the script will terminate with an exit status of 1.
Si el script requiere la temperatura del directorio y no existe, el script terminará con un estado de salida de 1.

## Summing Up
## Resumen

We started this chapter with a question. How could we make our sys_info_page script detect whether the user had permission to read all the home directories? With our knowledge of if , we can solve the problem by adding this code to the **report_home_space** function:
Comenzamos este capítulo con una pregunta. ¿Cómo podemos hacer que nuestro script sys_info_page detecte si el usuario tiene permiso para leer todos los directorios de inicio? Con nuestro conocimiento de if, podemos resolver el problema agregando este código a la función **report_home_space**:

---

```
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
}
```

---

We evaluate the output of the id command. With the -u option, id outputs the numeric user ID number of the effective user. The superuser is always ID zero, and every other user is a number greater than zero. Knowing this, we can construct two different here documents, one taking advantage of superuser privileges and the other restricted to the user's own home directory.
Evaluamos la salida del comando id. Con la opción -u, id genera el número de identificación de usuario numérico del usuario efectivo. El superusuario es siempre ID cero y todos los demás usuarios son un número mayor que cero. Sabiendo esto, podemos construir dos documentos aquí diferentes, uno aprovechando los privilegios de superusuario y el otro restringido al directorio personal del usuario.

We are going to take a break from the sys_info_page program, but don't worry. It will be back. In the meantime, we'll cover some topics that we'll need when we resume our work.
Vamos a tomarnos un descanso del programa sys_info_page, pero no se preocupe. Volverá. Mientras tanto, cubriremos algunos temas que necesitaremos cuando reanudemos nuestro trabajo.