34

string and Numbers
Cadenas y Números

Computer programs are all about working with data. In past chapters, we have focused on processing data at the file level. However, many programming problems need to be solved using smaller units of data such as strings and numbers.
Los programas de computadora tienen que ver con trabajar con datos. En capítulos anteriores, nos hemos centrado en procesar datos a nivel de archivo. Sin embargo, muchos problemas de programación deben resolverse utilizando unidades de datos más pequeñas, como cadenas y números.

In this chapter, we will look at several shell features that are used to manipulate strings and numbers. The shell provides a variety of parameter expansions that perform string operations. In addition to arithmetic expansion (which we touched upon in Chapter 7), there is a well-known command line program called bc , which performs higher-level math.
En este capítulo, veremos varias características del shell que se utilizan para manipular cadenas y números. El shell proporciona una variedad de expansiones de parámetros que realizan operaciones de cadena. Además de la expansión aritmética (que mencionamos en el Capítulo 7), existe un programa de línea de comandos muy conocido llamado bc, que realiza operaciones matemáticas de alto nivel.

Parameter Expansion
Expansión de parámetros

Though parameter expansion came up in Chapter 7, we did not cover it in detail because most parameter expansions are used in scripts rather than on the command line. We have already worked with some forms of parameter expansion, for example, shell variables. The shell provides many more.
Aunque la expansión de parámetros surgió en el Capítulo 7, no la cubrimos en detalle porque la mayoría de las expansiones de parámetros se utilizan en scripts en lugar de en la línea de comandos. Ya hemos trabajado con algunas formas de expansión de parámetros, por ejemplo, variables de shell. La cáscara proporciona muchos más.

Note
Nota

It's always good practice to enclose parameter expansions in double quotes to prevent unwanted word splitting, unless there is a specific reason not to. This is especially true when dealing with filenames since they can often include embedded spaces and other assorted nastiness.

> Siempre es una buena práctica incluir las expansiones de parámetros entre comillas dobles para evitar la división de palabras no deseadas, a menos que haya una razón específica para no hacerlo. Esto es especialmente cierto cuando se trata de nombres de archivos, ya que a menudo pueden incluir espacios incrustados y otras cosas desagradables.

Basic Parameters
Parametros basicos

The simplest form of parameter expansion is reflected in the ordinary use of variables. Here's an example:
La forma más simple de expansión de parámetros se refleja en el uso ordinario de variables. Aquí tienes un ejemplo:

```
$a
```

When expanded, this becomes whatever the variable a contains. Simple parameters may also be surrounded by braces.
Cuando se expande, se convierte en lo que contenga la variable a. Los parámetros simples también pueden estar rodeados de llaves.

```
${a}
```

This has no effect on the expansion but is required if the variable is adjacent to other text, which may confuse the shell. In this example, we attempt to create a filename by appending the string _file to the contents of the variable a :
Esto no tiene ningún efecto en la expansión, pero es necesario si la variable es adyacente a otro texto, lo que puede confundir al shell. En este ejemplo, intentamos crear un nombre de archivo agregando la cadena _file al contenido de la variable a:

```
[me@linuxbox ~]$ a="foo"
[me@linuxbox ~]$ echo "$a_file"
```

If we perform this sequence of commands, the result will be nothing because the shell will try to expand a variable named a_file rather than a. This problem can be solved by adding braces around the "real" variable name.
Si realizamos esta secuencia de comandos, el resultado no será nada porque el shell intentará expandir una variable llamada a_file en lugar de a. Este problema se puede resolver agregando llaves alrededor del nombre de la variable "real".

```
[me@linuxbox ~]$ echo "${a}_file"
foo_file
```

We have also seen that positional parameters greater than nine can be accessed by surrounding the number in braces. For example, to access the eleventh positional parameter, we can do this:
También hemos visto que se puede acceder a parámetros posicionales mayores de nueve rodeando el número entre llaves. Por ejemplo, para acceder al undécimo parámetro posicional, podemos hacer esto:

```
${11}
```

Expansions to Manage Empty Variables
Expansiones para administrar variables vacías

Several parameter expansions are intended to deal with nonexistent and empty variables. These expansions are handy for handling missing positional parameters and assigning default values to parameters. Here is one such expansion:
Varias expansiones de parámetros están pensadas para tratar con variables vacías e inexistentes. Estas expansiones son útiles para manejar los parámetros posicionales que faltan y asignar valores predeterminados a los parámetros. Aquí hay una de esas expansiones:

```
${parameter:-word}
```

If parameter is unset (i.e., does not exist) or is empty, this expansion results in the value of word . If parameter is not empty, the expansion results in the value of parameter.
Si el parámetro no está configurado (es decir, no existe) o está vacío, esta expansión da como resultado el valor de palabra. Si el parámetro no está vacío, la expansión da como resultado el valor del parámetro.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
substitute value if unset
[me@linuxbox ~]$ echo $foo

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
bar
[me@linuxbox ~]$ echo $foo
bar
```

Here is another expansion, in which we use the equal sign instead of a dash:
Aquí hay otra expansión, en la que usamos el signo igual en lugar de un guión:

```
${parameter:=word}
```

If parameter is unset or empty, this expansion results in the value of word.
Si el parámetro no está configurado o está vacío, esta expansión da como resultado el valor de palabra.

In addition, the value of word is assigned to parameter . If parameter is not empty, the expansion results in the value of parameter.
Además, el valor de la palabra se asigna al parámetro. Si el parámetro no está vacío, la expansión da como resultado el valor del parámetro.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
default value if unset
[me@linuxbox ~]$ echo $foo
default value if unset
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
bar
[me@linuxbox ~]$ echo $foo
bar
```

> # Note
> # Nota
>
> Positional and other special parameters cannot be assigned this way.
> Los parámetros posicionales y otros parámetros especiales no se pueden asignar de esta manera.

Here we use a question mark:
Aquí usamos un signo de interrogación:

```
${parameter:?word}
```

If parameter is unset or empty, this expansion causes the script to exit with an error, and the contents of word are sent to standard error. If parameter is not empty, the expansion results in the value of parameter.
Si el parámetro no está configurado o está vacío, esta expansión hace que el script se cierre con un error y el contenido de la palabra se envía al error estándar. Si el parámetro no está vacío, la expansión da como resultado el valor del parámetro.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bash: foo: parameter is empty
[me@linuxbox ~]$ echo $?
1
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bar
[me@linuxbox ~]$ echo $?
0
```

Here we use a plus sign:
Aquí usamos un signo más:

```
${parameter:+word}
```

If parameter is unset or empty, the expansion results in nothing. If parameter is not empty, the value of word is substituted for parameter ; however, the value of parameter is not changed.
Si el parámetro no está configurado o está vacío, la expansión no da como resultado nada. Si el parámetro no está vacío, el valor de la palabra se sustituye por el parámetro; sin embargo, el valor del parámetro no cambia.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}
substitute value if set
```

Expansions That Return Variable Names
Expansiones que devuelven nombres de variables

The shell has the ability to return the names of variables. This is used in some rather exotic situations.
El shell tiene la capacidad de devolver los nombres de las variables. Esto se usa en algunas situaciones bastante exóticas.

```
${!prefix*}
${!prefix@}
```

This expansion returns the names of existing variables with names beginning with prefix . According to the bash documentation, both forms of the expansion perform identically. Here, we list all the variables in the environment with names that begin with BASH:
Esta expansión devuelve los nombres de las variables existentes con nombres que comienzan con prefijo. De acuerdo con la documentación de bash, ambas formas de expansión funcionan de manera idéntica. Aquí, enumeramos todas las variables en el entorno con nombres que comienzan con BASH:

```
[me@linuxbox ~]$ echo ${!BASH*}
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_COMPLETION BASH_COMPLETION_DIR
BASH_LINENO BASH_SOURCE BASH_SUBSHELL BASH_VERSINFO BASH_VERSION
```

String Operations
Operaciones de cadena

There is a large set of expansions that can be used to operate on strings. Many of these expansions are particularly well suited for operations on pathnames. The following expansion:
Existe un gran conjunto de expansiones que se pueden utilizar para operar con cuerdas. Muchas de estas expansiones son especialmente adecuadas para operaciones en nombres de ruta. La siguiente expansión:

```
${#parameter}
```

expands into the length of the string contained by parameter . Normally, parameter is a string; however, if parameter is either @ or * , then the expansion results in the number of positional parameters.
se expande a la longitud de la cadena contenida por parámetro. Normalmente, el parámetro es una cadena; sin embargo, si el parámetro es @ o *, entonces la expansión da como resultado el número de parámetros posicionales.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo "'$foo' is ${#foo} characters long."
'This string is long.' is 20 characters long.
```

The following expansions are used to extract a portion of the string contained in **parameter** :
Las siguientes expansiones se utilizan para extraer una parte de la cadena contenida en **parámetro**:

```
${parameter:offset}
${parameter:offset:length}
```

The extraction begins at offset characters from the beginning of the string and continues until the end of the string, unless length is specified.
La extracción comienza en los caracteres de desplazamiento desde el principio de la cadena y continúa hasta el final de la cadena, a menos que se especifique la longitud.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo:5}
string is long.
[me@linuxbox ~]$ echo ${foo:5:6}
string
```

If the value of offset is negative, it is taken to mean it starts from the end of the string rather than the beginning. Note that negative values must be preceded by a space to prevent confusion with the ${parameter:-word} expansion. length , if present, must not be less than zero.
Si el valor de desplazamiento es negativo, se considera que comienza desde el final de la cadena en lugar del principio. Tenga en cuenta que los valores negativos deben ir precedidos de un espacio para evitar confusiones con la expansión $ {parámetro: -word}. la longitud, si está presente, no debe ser menor que cero.

If **parameter** is @ , the result of the expansion is length positional parameters, starting at offset .
Si **parámetro** es @, el resultado de la expansión son los parámetros posicionales de longitud, comenzando en el desplazamiento.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo: -5}
long.
[me@linuxbox ~]$ echo ${foo: -5:2}
lo
```

The following expansions remove a leading portion of the string contained in parameter defined by pattern.
Las siguientes expansiones eliminan una parte inicial de la cadena contenida en el parámetro definido por patrón.

```
${parameter#pattern}
${parameter##pattern}
```

---

pattern is a wildcard pattern like those used in pathname expansion. The difference in the two forms is that the # form removes the shortest match, while the ## form removes the longest match.

patrón es un patrón comodín como los que se utilizan en la expansión de nombre de ruta. La diferencia entre las dos formas es que la forma # elimina la coincidencia más corta, mientras que la forma ## elimina la coincidencia más larga.

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo#*.}
txt.zip
[me@linuxbox ~]$ echo ${foo##*.}
zip
```

The following are the same as the previous # and ## expansions, except they remove text from the end of the string contained in parameter rather than from the beginning.
Las siguientes son las mismas que las expansiones # y ## anteriores, excepto que eliminan el texto del final de la cadena contenida en el parámetro en lugar de hacerlo desde el principio.

```
${parameter%pattern}
${parameter%%pattern}
```

Here is an example:
aquí está un ejemplo:

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo%.*}
file.txt
[me@linuxbox ~]$ echo ${foo%%.*}
file
```

The following expansions perform a search-and-replace operation upon the contents of parameter:
Las siguientes expansiones realizan una operación de búsqueda y reemplazo sobre el contenido del parámetro:

```
${parameter/pattern/string}
${parameter//pattern/string}
${parameter/#pattern/string}
${parameter/%pattern/string}
```

If text is found matching wildcard pattern , it is replaced with the contents of string . In the normal form, only the first occurrence of pattern is replaced. In the // form, all occurrences are replaced. The /# form requires that the match occur at the beginning of the string, and the /% form requires the match to occur at the end of the string. In every form, /string may be omitted, causing the text matched by pattern to be deleted.
Si se encuentra texto que coincide con el patrón de comodín, se reemplaza con el contenido de la cadena. En la forma normal, solo se reemplaza la primera aparición de patrón. En el formulario //, se reemplazan todas las apariciones. La forma /# requiere que la coincidencia ocurra al principio de la cadena, y la forma /% requiere que la coincidencia ocurra al final de la cadena. En todas las formas, se puede omitir /sting, lo que provoca que se elimine el texto que coincide con el patrón.

```
[me@linuxbox~]$ foo=JPG.JPG
[me@linuxbox~]$ echo ${foo/JPG/jpg}
jpg.JPG
[me@linuxbox~]$ echo ${foo//JPG/jpg}
jpg.jpg
[me@linuxbox~]$ echo ${foo/#JPG/jpg}
jpg.JPG
[me@linuxbox~]$ echo ${foo/%JPG/jpg}
JPG.jpg
```

Parameter expansion is a good thing to know. The string manipulation expansions can be used as substitutes for other common commands such as sed and cut . Expansions can improve the efficiency of scripts by eliminating the use of external programs. As an example, we will modify the longest-word program discussed in the previous chapter to use the parameter expansion ${#j} in place of the command substitution $(echo -n $j | wc -c) and its resulting subshell, like so:
Es bueno saber la expansión de parámetros. Las expansiones de manipulación de cadenas se pueden utilizar como sustitutos de otros comandos comunes como sed y cut. Las expansiones pueden mejorar la eficiencia de los scripts al eliminar el uso de programas externos. Como ejemplo, modificaremos el programa de la palabra más larga discutido en el capítulo anterior para usar el parámetro de expansión ${#j} en lugar de la sustitución del comando $(echo -n $j | wc -c) y su subshell resultante, al igual que:

```
#!/bin/bash

# longest-word3: find longest string in a file

for i; do
    if [[ -r "$i" ]]; then
        max_word=
        max_len=0
        for j in $(strings $i); do
            len="${#j}"
            if (( len > max_len )); then
                max_len="$len"
                max_word="$j"
```

```
            fi
        done
        echo "$i: '$max_word' ($max_len characters)"
    fi
done
```

Next, we will compare the efficiency of the two versions by using the time command.
A continuación, compararemos la eficiencia de las dos versiones usando el comando time.

```
[me@linuxbox ~]$ time longest-word2 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)

real    0m3.618s
user    0m1.544s
sys     0m1.768s
[me@linuxbox ~]$ time longest-word3 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)

real    0m0.060s
user    0m0.056s
sys     0m0.008s
```

The original version of the script takes 3.618 seconds to scan the text file, while the new version, using parameter expansion, takes only 0.06 seconds, which is a significant improvement.
La versión original del script tarda 3.618 segundos en escanear el archivo de texto, mientras que la nueva versión, que usa la expansión de parámetros, toma solo 0.06 segundos, lo cual es una mejora significativa.

Case Conversion
Conversión de casos

---

bash has four parameter expansions and two declare command options to support the uppercase/lowercase conversion of strings.
bash tiene cuatro expansiones de parámetros y dos opciones de comando de declaración para admitir la conversión de cadenas de mayúsculas/minúsculas.

So, what is case conversion good for? Aside from the obvious aesthetic value, it has an important role in programming. Let's consider the case of a database lookup. Imagine that a user has entered a string into a data input field that we want to look up in a database. It's possible the user will enter the value in all uppercase letters or lowercase letters or a combination of both. We certainly don't want to populate our database with every possible permutation of uppercase and lowercase spellings. What to do?
Entonces, ¿para qué sirve la conversión de casos? Aparte del valor estético obvio, tiene un papel importante

en la programación. Consideremos el caso de una búsqueda en una base de datos. Imagine que un usuario ha ingresado una cadena en un campo de entrada de datos que queremos buscar en una base de datos. Es posible que el usuario ingrese el valor en letras mayúsculas o minúsculas o una combinación de ambas. Ciertamente, no queremos poblar nuestra base de datos con todas las posibles permutaciones de ortografía en mayúsculas y minúsculas. ¿Qué hacer?

A common approach to this problem is to normalize the user's input. That is, convert it into a standardized form before we attempt the database lookup. We can do this by converting all the characters in the user's input to either lower or uppercase and ensure that the database entries are normalized the same way.
Un enfoque común para este problema es normalizar la entrada del usuario. Es decir, conviértalo en un formulario estandarizado antes de intentar la búsqueda en la base de datos. Podemos hacer esto convirtiendo todos los caracteres en la entrada del usuario a minúsculas o mayúsculas y asegurarnos de que las entradas de la base de datos estén normalizadas de la misma manera.

The declare command can be used to normalize strings to either uppercase or lowercase. Using declare , we can force a variable to always contain the desired format no matter what is assigned to it.
El comando declare puede usarse para normalizar cadenas a mayúsculas o minúsculas. Usando declare, podemos forzar que una variable contenga siempre el formato deseado sin importar lo que se le asigne.

```
#!/bin/bash

# ul-declare: demonstrate case conversion via declare

declare -u upper
declare -l lower
if [[ $1 ]]; then
    upper="$1"
    lower="$1"
    echo "$upper"
    echo "$lower"
fi
```

In the preceding script, we use declare to create two variables, upper and lower . We assign the value of the first command line argument (positional parameter 1) to each of the variables and then display them on the screen.
En el script anterior, usamos declare para crear dos variables, superior e inferior. Asignamos el valor del primer argumento de la línea de comando (parámetro posicional 1) a cada una de las variables y luego las mostramos en la pantalla.

```
[me@linuxbox ~]$ ul-declare aBc
ABC
abc
```

As we can see, the command line argument (aBc) has been normalized. In addition to declare , there are four parameter expansions that perform upper/lowercase conversion, as described in Table 34-1.
Como podemos ver, el argumento de la línea de comandos (aBc) se ha normalizado. Además de declarar, hay cuatro expansiones de parámetros que realizan la conversión de mayúsculas / minúsculas, como se describe en la Tabla 34-1.

> Table 34-1: Case Conversion Parameter Expansions
> Tabla 34-1: Expansiones de parámetros de conversión de casos

Format	Result
${ parameter,,pattern }	Expand the value of parameter into all lowercase. pattern is an optional shell pattern that will limit which characters (for example, [A-F]) will be converted. See the bash man page for a full description of patterns. Expanda el valor del parámetro en minúsculas. patrón es un patrón de shell opcional que limitará qué caracteres (por ejemplo, [A-F]) se convertirán. Consulte la página del manual de bash para obtener una descripción completa de los patrones.
${ parameter,pattern }	Expand the value of parameter , changing only the first character to lowercase. Expanda el valor del parámetro, cambiando solo el primer carácter a minúsculas.
${ parameter^^pattern }	Expand the value of parameter into all uppercase letters. Expanda el valor del parámetro en todas las letras mayúsculas.
${ parameter^pattern }	Expand the value of parameter , changing only the first character to uppercase (capitalization). Expanda el valor del parámetro, cambiando solo el primer carácter a mayúsculas (mayúsculas).

Here is a script that demonstrates these expansions:
Aquí hay un guión que demuestra estas expansiones:

```
#!/bin/bash

# ul-param: demonstrate case conversion via parameter expansion

if [[ "$1" ]]; then
    echo "${1,,}"
    echo "${1,}"
    echo "${1^^}"
    echo "${1^}"
fi
```

Here is the script in action:
Aquí está el guión en acción:

```
[me@linuxbox ~]$ ul-param aBc
abc
aBc
ABC
ABc
```

Again, we process the first command line argument and output the four variations supported by the parameter expansions. While this script uses the first positional parameter, parameter may be any string, variable, or string expression.
Nuevamente, procesamos el primer argumento de la línea de comando y mostramos las cuatro variaciones admitidas por las expansiones de parámetros. Si bien este script usa el primer parámetro posicional, el parámetro puede ser cualquier cadena, variable o expresión de cadena.

Arithmetic Evaluation and Expansion
Evaluación y expansión aritmética

We looked at arithmetic expansion in Chapter 7. It is used to perform various arithmetic operations on integers. Its basic form is as follows:
Examinamos la expansión aritmética en el capítulo 7. Se utiliza para realizar varias operaciones aritméticas con números enteros. Su forma básica es la siguiente:

```
$((expression))
```

where expression is a valid arithmetic expression.
donde expresión es una expresión aritmética válida.

This is related to the compound command (( )) used for arithmetic evaluation (truth tests) we encountered in Chapter 27.
Esto está relacionado con el comando compuesto (()) utilizado para la evaluación aritmética (pruebas de verdad) que encontramos en el Capítulo 27.

In previous chapters, we saw some of the common types of expressions and operators. Here, we will look at a more complete list.
En capítulos anteriores, vimos algunos de los tipos comunes de expresiones y operadores. Aquí, veremos una lista más completa.

Number Bases
Bases numéricas

In Chapter 9, we got a look at octal (base 8) and hexadecimal (base 16) numbers. In arithmetic expressions, the shell supports integer constants in any base. Table 34-2 shows the notations used to specify the bases.
En el Capítulo 9, echamos un vistazo a los números octales (base 8) y hexadecimales (base 16). En

expresiones aritméticas, el shell admite constantes enteras en cualquier base. La Tabla 34-2 muestra las notaciones utilizadas para especificar las bases.

Table 34-2: Specifying Different Number Bases
Tabla 34-2: Especificación de bases numéricas diferentes

Notation	Description
number	By default, numbers without any notation are treated as decimal (base 10) integers. De forma predeterminada, los números sin notación se tratan como enteros decimales (base 10).
0number	In arithmetic expressions, numbers with a leading zero are considered octal. En expresiones aritméticas, los números con un cero a la izquierda se consideran octales.
0xnumber	Hexadecimal notation. Notación hexadecimal.
base#number	number is in base. el número está en la base.

Here are some examples:
Aquí hay unos ejemplos:

```
[me@linuxbox ~]$ echo $((0xff))
255
[me@linuxbox ~]$ echo $((2#11111111))
255
```

In the previous examples, we print the value of the hexadecimal number ff (the largest two-digit number) and the largest eight-digit binary (base 2) number.
En los ejemplos anteriores, imprimimos el valor del número hexadecimal ff (el número más grande de dos dígitos) y el número binario de ocho dígitos más grande (base 2).

Unary Operators
Operadores unarios

There are two unary operators, + and - , which are used to indicate whether a number is positive or negative, respectively. An example is -5.
Hay dos operadores unarios, + y -, que se utilizan para indicar si un número es positivo o negativo, respectivamente. Un ejemplo es -5.

Simple Arithmetic
Aritmética simple

Table 34-3 lists the ordinary arithmetic operators.
La tabla 34-3 enumera los operadores aritméticos ordinarios.

> Table 34-3: Arithmetic Operators
> Tabla 34-3: Operadores aritméticos

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Integer division
**	Exponentiation
%	Modulo (remainder)

Most of these are self-explanatory, but integer division and modulo require further discussion.
La mayoría de ellos se explican por sí mismos, pero la división de enteros y el módulo requieren más discusión.

Since the shell's arithmetic operates only on integers, the results of division are always whole numbers.
Dado que la aritmética del shell opera solo con números enteros, los resultados de la división son siempre números enteros.

```
[me@linuxbox ~]$ echo $(( 5 / 2 ))
2
```

This makes the determination of a remainder in a division operation more important.
Esto hace que la determinación de un resto en una operación de división sea más importante.

```
[me@linuxbox ~]$ echo $(( 5 % 2 ))
1
```

By using the division and modulo operators, we can determine that 5 divided by 2 results in 2, with a remainder of 1.

Al usar los operadores de división y módulo, podemos determinar que 5 dividido por 2 da como resultado 2, con un resto de 1.

Calculating the remainder is useful in loops. It allows an operation to be performed at specified intervals during the loop's execution. In the following example, we display a line of numbers, highlighting each multiple of 5:
Calcular el resto es útil en bucles. Permite que se realice una operación a intervalos específicos durante la ejecución del bucle. En el siguiente ejemplo, mostramos una línea de números, resaltando cada múltiplo de 5:

```
#!/bin/bash

# modulo: demonstrate the modulo operator

for ((i = 0; i <= 20; i = i + 1)); do
    remainder=$((i % 5))
    if (( remainder == 0 )); then
        printf "<%d> " "$i"
    else
        printf "%d " "$i"
    fi
done
printf "\n"
```

When executed, the results look like this:
Cuando se ejecuta, los resultados se ven así:

```
[me@linuxbox ~]$ modulo
<0> 1 2 3 4 <5> 6 7 8 9 <10> 11 12 13 14 <15> 16 17 18 19 <20>
```

Assignment
Asignación

Although its uses may not be immediately apparent, arithmetic expressions may perform assignment. We have performed assignment many times, though in a different context. Each time we give a variable a value, we are performing assignment. We can also do it within arithmetic expressions.
Aunque sus usos pueden no ser evidentes de inmediato, las expresiones aritméticas pueden realizar asignaciones. Hemos realizado asignaciones muchas veces, aunque en un contexto diferente. Cada vez que le damos un valor a una variable, estamos realizando una asignación. También podemos hacerlo dentro de expresiones aritméticas.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo $foo
[me@linuxbox ~]$ if (( foo = 5 )); then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ echo $foo
5
```

In the preceding example, we first assign an empty value to the variable foo and verify that it is indeed empty. Next, we perform an if with the compound command ((foo = 5)) . This process does two interesting things: it assigns the value of 5 to the variable foo , and it evaluates to true because foo was assigned a non-zero value.
En el ejemplo anterior, primero asignamos un valor vacío a la variable foo y verificamos que de hecho esté vacío. A continuación, realizamos un if con el comando compuesto ((foo = 5)). Este proceso hace dos cosas interesantes: asigna el valor de 5 a la variable foo y evalúa como verdadero porque a foo se le asignó un valor distinto de cero.

> ## Note
> ## Nota
>
> It is important to remember the exact meaning of = in the previous expression. A single = performs assignment. foo = 5 says “make foo equal to 5,” while == evaluates equivalence. foo == 5 says “does foo equal 5?” This is a common feature in many programming languages. In the shell, this can be a little confusing because the test command accepts a single = for string equivalence. This is yet another reason to use the more modern [[ ]] and (( )) compound commands in place of test .
> Es importante recordar el significado exacto de = en la expresión anterior. Un solo = realiza la tarea. foo = 5 dice "hacer foo igual a 5", mientras que == evalúa la equivalencia. foo == 5 dice "¿foo es igual a 5?" Esta es una característica común en muchos lenguajes de programación. En el shell, esto puede ser un poco confuso porque el comando de prueba acepta un solo = para la equivalencia de cadenas. Esta es otra razón más para usar los comandos compuestos [[]] y (()) más modernos en lugar de test.

In addition to the = notation, the shell also provides notations that perform some very useful assignments, as described in Table 34-4.
Además de la notación =, el shell también proporciona notaciones que realizan algunas asignaciones muy útiles, como se describe en la Tabla 34-4.

> Table 34-4: Assignment Operators
> Tabla 34-4: Operadores de asignación

Notation	Description
parameter = value	Simple assignment. Assigns value to parameter . Asignación sencilla. Asigna valor al parámetro.
parameter += value	Addition. Equivalent to parameter = parameter + value . Adición. Equivalente a parámetro = parámetro + valor.

Notation	Description
parameter -= value	Subtraction. Equivalent to parameter = parameter – value . Sustracción. Equivalente a parámetro = parámetro - valor.
parameter *= value	Multiplication. Equivalent to parameter = parameter * value . Multiplicación. Equivalente a parámetro = parámetro * valor.
parameter /= value	Integer division. Equivalent to parameter = parameter / value . División entera. Equivalente a parámetro = parámetro / valor.
parameter %= value	Modulo. Equivalent to parameter = parameter % value . Modulo. Equivalente a parámetro = parámetro% valor.
parameter++	Variable post-increment. Equivalent to parameter = parameter + 1 (however, see the following discussion). Post-incremento variable. Equivalente a parámetro = parámetro + 1 (sin embargo, consulte la siguiente discusión).
parameter––	Variable post-decrement. Equivalent to parameter = parameter – 1. Post-decremento variable. Equivalente a parámetro = parámetro - 1.
++parameter	Variable pre-increment. Equivalent to parameter = parameter + 1. Preincremento variable. Equivalente a parámetro = parámetro + 1.
--parameter	Variable pre-decrement. Equivalent to parameter = parameter – 1. Pre-decremento variable. Equivalente a parámetro = parámetro - 1.

These assignment operators provide a convenient shorthand for many common arithmetic tasks. Of special interest are the increment (++) and decrement (––) operators, which increase or decrease the value of their parameters by one. This style of notation is taken from the C programming language and has been incorporated into a number of other programming languages, including bash.
Estos operadores de asignación proporcionan un método abreviado conveniente para muchas tareas aritméticas comunes. De especial interés son los operadores de incremento (++) y decremento (––), que aumentan o disminuyen el valor de sus parámetros en uno. Este estilo de notación se toma del lenguaje de programación C y se ha incorporado a varios otros lenguajes de programación, incluido bash.

The operators may appear either at the front of a parameter or at the end. While they both either increment or decrement the parameter by one, the two placements have a subtle difference. If placed at the front of the parameter, the parameter is incremented (or decremented) before the parameter is returned. If placed after, the operation is performed after the parameter is returned. This is rather strange, but it is the intended behavior. Here is a demonstration:
Los operadores pueden aparecer al principio de un parámetro o al final. Si bien ambos aumentan o disminuyen el parámetro en uno, las dos ubicaciones tienen una diferencia sutil. Si se coloca al principio del parámetro, el parámetro se incrementa (o decrementa) antes de que se devuelva el parámetro. Si se coloca después, la operación se realiza después de que se devuelva el parámetro. Esto es bastante extraño, pero es el comportamiento previsto. Aquí hay una demostración:

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((foo++))
1
[me@linuxbox ~]$ echo $foo
2
```

If we assign the value of one to the variable foo and then increment it with the ++ operator placed after the parameter name, foo is returned with the value of one. However, if we look at the value of the variable a second time, we see the incremented value. If we place the ++ operator in front of the parameter, we get the more expected behavior.
Si asignamos el valor de uno a la variable foo y luego lo incrementamos con el operador ++ colocado después del nombre del parámetro, foo se devuelve con el valor de uno. Sin embargo, si miramos el valor de la variable por segunda vez, vemos el valor incrementado. Si colocamos el operador ++ delante del parámetro, obtenemos el comportamiento más esperado.

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((++foo))
2
[me@linuxbox ~]$ echo $foo
2
```

For most shell applications, prefixing the operator will be the most useful.
Para la mayoría de las aplicaciones de shell, el prefijo del operador será lo más útil.

The ++ and -- operators are often used in conjunction with loops. We will make some improvements to our modulo script to tighten it up a bit.
Los operadores ++ y - se utilizan a menudo junto con bucles. Haremos algunas mejoras en nuestro script de módulo para ajustarlo un poco.

```
#!/bin/bash

# modulo2: demonstrate the modulo operator

for ((i = 0; i <= 20; ++i )); do
    if (((i % 5) == 0 )); then
        printf "<%d> " "$i"
    else
        printf "%d " "$i"
    fi
done
printf "\n"
```

Bit Operations
Operaciones de bits

One class of operators manipulates numbers in an unusual way. These operators work at the bit level. They are used for certain kinds of low-level tasks, often involving setting or reading bit flags (see Table 34-5).
Una clase de operadores manipula los números de una manera inusual. Estos operadores trabajan a nivel de bits. Se utilizan para ciertos tipos de tareas de bajo nivel, que a menudo implican establecer o leer indicadores de bits (consulte la Tabla 34-5).

> Table 34-5: Bit Operators
> Tabla 34-5: Operadores de bits

Operator	Description
~	Bitwise negation. Negate all the bits in a number. Negación bit a bit. Niega todos los bits de un número.
<<	Left bitwise shift. Shift all the bits in a number to the left. Desplazamiento bit a la izquierda. Mueva todos los bits de un número a la izquierda.
>>	Right bitwise shift. Shift all the bits in a number to the right. Desplazamiento bit a bit a la derecha. Mueva todos los bits de un número a la derecha.
&	Bitwise AND. Perform an AND operation on all the bits in two numbers.
|	Bitwise OR. Perform an OR operation on all the bits in two numbers. OR bit a bit. Realice una operación OR en todos los bits en dos números.
^	Bitwise XOR. Perform an exclusive OR operation on all the bits in two numbers. XOR bit a bit. Realice una operación OR exclusiva en todos los bits en dos números.

Note that there are also corresponding assignment operators (for example, <<=) for all but bitwise negation.
Tenga en cuenta que también hay operadores de asignación correspondientes (por ejemplo, << =) para todos menos la negación bit a bit.

Here we will demonstrate producing a list of powers of 2, using the left bitwise shift operator:
Aquí demostraremos cómo producir una lista de potencias de 2, usando el operador de desplazamiento bit a bit a la izquierda:

```
[me@linuxbox ~]$ for ((i=0;i<8;++i)); do echo $((1<<i)); done
1
2
4
8
16
32
64
128
```

Logic
Lógica

As we discovered in Chapter 27, the (( )) compound command supports a variety of comparison operators. There are a few more that can be used to evaluate logic. Table 34-6 provides the complete list.
Como descubrimos en el Capítulo 27, el comando compuesto (()) admite una variedad de operadores de comparación. Hay algunos más que se pueden usar para evaluar la lógica. La Tabla 34-6 proporciona la lista completa.

> Table 34-6: Comparison Operators
> Tabla 34-6: Operadores de comparación

Operator	Description
<=	Less than or equal to. Menos que o igual a.
>=	Greater than or equal to. Mayor qué o igual a.
<	Less than. Menos que.
>	Greater than. Mas grande que.
==	Equal to. Igual a.
!=	Not equal to. No igual a
&&	Logical AND. AND logico
expr1? expr2:expr3	Comparison (ternary) operator. If expression expr1 evaluates tobe nonzero (arithmetic true), then expr2 ; else expr3. Operador de comparación (ternario). Si la expresión expr1 se evalúa como distinta de cero (aritmética verdadera), entonces expr2; más expr3.

When used for logical operations, expressions follow the rules of arithmetic logic; that is, expressions that evaluate as zero are considered false, while non-zero expressions are considered true. The (( )) compound command maps the results into the shell's normal exit codes.
Cuando se utilizan para operaciones lógicas, las expresiones siguen las reglas de la lógica aritmética; es decir, las expresiones que se evalúan como cero se consideran falsas, mientras que las expresiones distintas de cero se consideran verdaderas. El comando compuesto (()) asigna los resultados a los códigos de salida normales del shell.

```
[me@linuxbox ~]$ if ((1)); then echo "true"; else echo "false"; fi
true
[me@linuxbox ~]$ if ((0)); then echo "true"; else echo "false"; fi
false
```

The strangest of the logical operators is the ternary operator. This operator (which is modeled after the one in the C programming language) performs a stand-alone logical test. It can be used as a kind of if / then / else statement. It acts on three arithmetic expressions (strings won't work), and if the first expression is true (or non-zero), the second expression is performed. Otherwise, the third expression is performed. We can try this on the command line:
El más extraño de los operadores lógicos es el operador ternario. Este operador (que sigue el modelo del lenguaje de programación C) realiza una prueba lógica independiente. Se puede utilizar como una especie de declaración if / then / else. Actúa sobre tres expresiones aritméticas (las cadenas no funcionan), y si la primera expresión es verdadera (o distinta de cero), se realiza la segunda expresión. De lo contrario, se realiza la tercera expresión. Podemos probar esto en la línea de comando:

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
1
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
0
```

Here we see a ternary operator in action. This example implements a toggle. Each time the operator is performed, the value of the variable a switches from zero to one or vice versa.
Aquí vemos un operador ternario en acción. Este ejemplo implementa un conmutador. Cada vez que se realiza el operador, el valor de la variable a cambia de cero a uno o viceversa.

Please note that performing assignment within the expressions is not straightforward. When attempted, bash will declare an error.
Tenga en cuenta que realizar asignaciones dentro de las expresiones no es sencillo. Cuando se intenta, bash declarará un error.

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?a+=1:a-=1))
bash: ((: a<1?a+=1:a-=1: attempted assignment to non-variable (error token
is "-=1")
```

This problem can be mitigated by surrounding the assignment expression with parentheses.
Este problema se puede mitigar rodeando la expresión de asignación entre paréntesis.

```
[me@linuxbox ~]$ ((a<1?(a+=1):(a-=1)))
```

Next is a more complete example of using arithmetic operators in a script that produces a simple table of numbers:
A continuación, se muestra un ejemplo más completo del uso de operadores aritméticos en un script que produce una tabla simple de números:

```
#!/bin/bash

# arith-loop: script to demonstrate arithmetic operators

finished=0
a=0
printf "a\ta**2\ta**3\n"
printf "=\t====\t====\n"

until ((finished)); do
    b=$((a**2))
    c=$((a**3))
    printf "%d\t%d\t%d\n" "$a" "$b" "$c"
    ((a<10?++a:(finished=1)))
done
```

In this script, we implement an until loop based on the value of the finished variable. Initially, the variable is set to zero (arithmetic false), and we continue the loop until it becomes non-zero. Within the loop, we calculate the square and cube of the counter variable a . At the end of the loop, the value of the counter variable is evaluated. If it is less than 10 (the maximum number of iterations), it is incremented by one, or else the variable finished is given the value of one, making finished arithmetically true, thereby terminating the loop. Running the script gives this result:
En este script, implementamos un bucle hasta basado en el valor de la variable terminada. Inicialmente, la variable se establece en cero (aritmética falsa) y continuamos el ciclo hasta que se vuelve diferente de cero. Dentro del ciclo, calculamos el cuadrado y el cubo de la variable de contador a. Al final del ciclo, se evalúa el valor de la variable del contador. Si es menor que 10 (el número máximo de iteraciones), se incrementa en uno, o de lo contrario a la variable finalizada se le da el valor de uno, haciendo que finalizado sea aritméticamente verdadero, terminando así el ciclo. Ejecutar el script da este resultado:

```
[me@linuxbox ~]$ arith-loop
a   a**2    a**3
=   ====    ====
0   0       0
```

```
1   1       1
2   4       8
3   9       27
..  ..      ..
```

bc - An Arbitrary Precision Calculator Language
bc - un lenguaje de calculadora de precisión arbitraria

We have seen how the shell can handle many types of integer arithmetic, but what if we need to perform higher math or even just use floating-point numbers? The answer is, we can't. At least not directly with the shell. To do this, we need to use an external program. There are several approaches we can take. Embedding Perl or AWK programs is one possible solution, but unfortunately, it's outside the scope of this book.
Hemos visto cómo el shell puede manejar muchos tipos de aritmética de enteros, pero ¿qué pasa si necesitamos realizar operaciones matemáticas superiores o incluso usar números de punto flotante? La respuesta es que no podemos. Al menos no directamente con el caparazón. Para hacer esto, necesitamos usar un programa externo. Hay varios enfoques que podemos adoptar. Incrustar programas Perl o AWK es una posible solución, pero desafortunadamente, está fuera del alcance de este libro.

Another approach is to use a specialized calculator program. One such program found on many Linux systems is called bc.
Otro enfoque es utilizar un programa de calculadora especializado. Uno de estos programas que se encuentra en muchos sistemas Linux se llama bc.

The bc program reads a file written in its own C-like language and executes it. A bc script may be a separate file, or it may be read from standard input. The bc language supports quite a few features including variables, loops, and programmer-defined functions. We won't cover bc entirely here, just enough to get a taste. bc is well documented by its man page.
El programa bc lee un archivo escrito en su propio lenguaje similar a C y lo ejecuta. Un script bc puede ser un archivo separado o puede leerse desde una entrada estándar. El lenguaje bc admite bastantes características, incluidas variables, bucles y funciones definidas por el programador. No cubriremos bc por completo aquí, solo lo suficiente para probarlo. bc está bien documentado por su página de manual.

Let's start with a simple example. We'll write a bc script to add 2 plus 2.
Comencemos con un ejemplo simple. Escribiremos un script bc para agregar 2 más 2.

```
/* A very simple bc script */
2 + 2
```

The first line of the script is a comment. bc uses the same syntax for comments as the C programming language. Comments, which may span multiple lines, begin with /* and end with */.
La primera línea del guión es un comentario. bc usa la misma sintaxis para los comentarios que el lenguaje

de programación C. Los comentarios, que pueden abarcar varias líneas, comienzan con / * y terminan con * /.

Using bc
Usando bc

If we save the previous bc script as foo.bc , we can run it this way:
Si guardamos el script bc anterior como foo.bc, podemos ejecutarlo de esta manera:

```
[me@linuxbox ~]$ bc foo.bc
bc 1.06.94
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation,
Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
4
```

If we look carefully, we can see the result at the very bottom, after the copyright message. This message can be suppressed with the -q (quiet) option.
Si miramos con atención, podemos ver el resultado en la parte inferior, después del mensaje de derechos de autor. Este mensaje se puede suprimir con la opción -q (silencioso).

bc can also be used interactively.
bc también se puede utilizar de forma interactiva

```
[me@linuxbox ~]$ bc -q
2 + 2
4
quit
```

When using bc interactively, we simply type the calculations we want to perform, and the results are immediately displayed. The bc command quit ends the interactive session.
Cuando usamos bc de forma interactiva, simplemente escribimos los cálculos que queremos realizar y los resultados se muestran inmediatamente. El comando bc quit finaliza la sesión interactiva.

It is also possible to pass a script to bc via standard input.
También es posible pasar un script a bc a través de una entrada estándar.

```
[me@linuxbox ~]$ bc < foo.bc
4
```

The ability to take standard input means that we can use here documents, here strings, and pipes to pass scripts. This is a here string example:
La capacidad de tomar entradas estándar significa que podemos usar aquí documentos, aquí cadenas y canalizaciones para pasar scripts. Este es un ejemplo de cadena aquí:

```
[me@linuxbox ~]$ bc <<< "2+2"
4
```

An Example Script
Un guion de ejemplo

As a real-world example, we will construct a script that performs a common calculation, monthly loan payments. In the script that follows, we use a here document to pass a script to bc:
Como ejemplo del mundo real, construiremos un script que realiza un cálculo común, pagos mensuales del préstamo. En el script que sigue, usamos un documento here para pasar un script a bc:

```
#!/bin/bash

# loan-calc: script to calculate monthly loan payments

PROGNAME="${0##*/}" # Use parameter expansion to get basename

usage () {
    cat <<- EOF
    Usage: $PROGNAME PRINCIPAL INTEREST MONTHS

    Where:

    PRINCIPAL is the amount of the loan.
    INTEREST is the APR as a number (7% = 0.07).
    MONTHS is the length of the loan's term.
EOF

}

if (($# != 3)); then
    usage
    exit 1
fi

principal=$1
interest=$2
months=$3
```

```
bc <<- EOF
    scale = 10
    i = $interest / 12
    p = $principal
    n = $months
    a = p * ((i * ((1 + i) ^ n)) / (((1 + i) ^ n) - 1))
    print a, "\n"
EOF
```

When executed, the results look like this:
Cuando se ejecuta, los resultados se ven así:

```
[me@linuxbox ~]$ loan-calc 135000 0.0775 180
1270.7222490000
```

This example calculates the monthly payment for a $135,000 loan at 7.75 percent APR for 180 months (15 years). Notice the precision of the answer. This is determined by the value given to the special scale variable in the bc script. A full description of the bc scripting language is provided by the bc man page. While its mathematical notation is slightly different from that of the shell (bc more closely resembles C), most of it will be quite familiar, based on what we have learned so far.
Este ejemplo calcula el pago mensual de un préstamo de $ 135,000 a una APR del 7.75 por ciento durante 180 meses (15 años). Note la precisión de la respuesta. Esto está determinado por el valor dado a la variable de escala especial en el script bc. La página de manual de bc proporciona una descripción completa del lenguaje de secuencias de comandos de bc. Si bien su notación matemática es ligeramente diferente a la del caparazón (bc se parece más a C), la mayor parte será bastante familiar, según lo que hemos aprendido hasta ahora.

Summing Up
Resumen

In this chapter, we learned about many of the little things that can be used to get the “real work” done in scripts. As our experience with scripting grows, the ability to effectively manipulate strings and numbers will prove extremely valuable. Our loan-calc script demonstrates that even simple scripts can be created to do some really useful things.
En este capítulo, aprendimos sobre muchas de las pequeñas cosas que se pueden usar para hacer el "trabajo real" en los scripts. A medida que crece nuestra experiencia con la creación de scripts, la capacidad de manipular de forma eficaz cadenas y números resultará extremadamente valiosa. Nuestro script credit-calc demuestra que incluso se pueden crear scripts simples para hacer cosas realmente útiles.

Extra Credit
Crédito adicional

While the basic functionality of the loan-calc script is in place, the script is far from complete. For extra credit, try improving the loan-calc script with the following features:
Si bien la funcionalidad básica de la secuencia de comandos de préstamo-calc está en su lugar, la secuencia de comandos está lejos de estar completa. Para obtener crédito adicional, intente mejorar la secuencia de comandos de préstamo-cálculo con las siguientes características:

- Full verification of the command line arguments
  Verificación completa de los argumentos de la línea de comandos
- A command line option to implement an “interactive” mode that will prompt the user to input the principal, interest rate, and term of the loan
  Una opción de línea de comandos para implementar un modo "interactivo" que solicitará al usuario que ingrese el capital, la tasa de interés y el plazo del préstamo.
- A better format for the output
  Un mejor formato para la salida