25

Starting a Proyect
Iniciar un proyecto

Starting with this chapter, we will begin to build a program. The purpose of this project is to see how various shell features are used to create programs and, more importantly, create good programs.
Comenzando con este capítulo, comenzaremos a construir un programa. El propósito de este proyecto es ver cómo se utilizan varias funciones del shell para crear programas y, lo que es más importante, crear buenos programas.

The program we will write is a report generator. It will present various statistics about our system and its status and will produce this report in HTML format so we can view it with a web browser such as Firefox or Chrome.
El programa que escribiremos es un generador de informes. Presentará varias estadísticas sobre nuestro sistema y su estado y producirá este informe en formato HTML para que podamos verlo con un navegador web como Firefox o Chrome.

Programs are usually built up in a series of stages, with each stage adding features and capabilities. The first stage of our program will produce a minimal HTML document that contains no system information.
Los programas generalmente se construyen en una serie de etapas, y cada etapa agrega características y capacidades. La primera etapa de nuestro programa producirá un documento HTML mínimo que no contiene información del sistema.

That will come later.
Eso vendrá después

First Stage: Minimal Document
Primera etapa: documento mínimo

The first thing we need to know is the format of a well-formed HTML document. It looks like this:
Lo primero que necesitamos saber es el formato de un documento HTML bien formado. Se parece a esto:

```
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        Page body.
```

```
    </body>
</html>
```

If we enter this into our text editor and save the file as foo.html, we can use the following URL in Firefox to view the file: file:///home/username/foo.html.
Si ingresamos esto en nuestro editor de texto y guardamos el archivo como foo.html, podemos usar la siguiente URL en Firefox para ver el archivo: file:///home/username/foo.html.

The first stage of our program will be able to output this HTML file to standard output. We can write a program to do this pretty easily. Let's start our text editor and create a new file named ~/bin/sys_info_page.
La primera etapa de nuestro programa podrá generar este archivo HTML en la salida estándar. Podemos escribir un programa para hacer esto con bastante facilidad. Iniciemos nuestro editor de texto y creemos un nuevo archivo llamado ~/bin/sys_info_page.

```
[~]$ vim ~/bin/sys_info_page
```

Enter the following program:
Ingrese al siguiente programa:

```
#!/bin/bash
# Program to output a system information page
echo "<html>"
echo "    <head>"
echo "        <title>Page Title</title>"
echo "    </head>"
echo "    <body>"
echo "        Page body."
echo "    </body>"
echo "</html>"
```

---

code 25-1

Our first attempt at this problem contains a shebang, a comment (always a good idea), and a sequence of echo commands, one for each line of output.
Nuestro primer intento de resolver este problema contiene un shebang, un comentario (siempre una buena idea) y una secuencia de comandos de eco, uno para cada línea de salida.

After saving the file, we'll make it executable and attempt to run it.
Después de guardar el archivo, lo haremos ejecutable e intentaremos ejecutarlo.

```
[~]$ chmod 755 ~/bin/sys_info_page
[~]$ sys_info_page
```

When the program runs, we should see the text of the HTML document displayed on the screen, because the echo commands in the script send their output to standard output. We'll run the program again and redirect the output of the program to the file sys_info_page.html so that we can view the result with a web browser.
Cuando el programa se ejecuta, deberíamos ver el texto del documento HTML mostrado en la pantalla, porque los comandos echo en el script envían su salida a la salida estándar. Ejecutaremos el programa nuevamente y redirigiremos la salida del programa al archivo sys_info_page.html para que podamos ver el resultado con un navegador web.

```
[~]$ sys_info_page > sys_info_page.html
[~]$ firefox sys_info_page.html
```

So far, so good.
Hasta ahora, todo bien.

When writing programs, it's always a good idea to strive for simplicity and clarity. Maintenance is easier when a program is easy to read and understand, not to mention that it can make the program easier to write by reducing the amount of typing. Our current version of the program works fine, but it could be simpler. We could actually combine all the echo commands into one, which will certainly make it easier to add more lines to the program's output. So, let's change our program to this:
Al escribir programas, siempre es una buena idea esforzarse por lograr la simplicidad y la claridad. El mantenimiento es más fácil cuando un programa es fácil de leer y comprender, sin mencionar que puede hacer que el programa sea más fácil de escribir al reducir la cantidad de escritura. Nuestra versión actual del programa funciona bien, pero podría ser más simple. De hecho, podríamos combinar todos los comandos de eco en uno, lo que sin duda hará que sea más fácil agregar más líneas a la salida del programa. Entonces, cambiemos nuestro programa a esto:

```
#!/bin/bash
# Program to output a system information page
echo "<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        Page body.
    </body>
</html>"
```

A quoted string may include newlines and, therefore, contain multiple lines of text. The shell will keep reading the text until it encounters the closing quotation mark. It works this way on the command line, too:
Una cadena entre comillas puede incluir nuevas líneas y, por lo tanto, contener varias líneas de texto. El shell seguirá leyendo el texto hasta que encuentre las comillas de cierre. También funciona de esta manera en la línea de comando:

```
[~]$ echo "<html>
>   <head>
>       <title>Page Title</title>
>   </head>
>   <body>
>       Page body.
>   </body>
> </html>"
```

The leading > character is the shell prompt contained in the PS2 shell variable. It appears whenever we type a multiline statement into the shell.
El carácter > inicial es el indicador de shell contenido en la variable de shell de PS2. Aparece cada vez que escribimos una declaración de varias líneas en el shell.

This feature is a little obscure right now, but later, when we cover multiline programming statements, it will turn out to be quite handy.
Esta característica es un poco oscura en este momento, pero más adelante, cuando cubramos las sentencias de programación de varias líneas, resultará bastante útil.

Second Stage: Adding a Little Data
Segunda etapa: agregar algunos datos

Now that our program can generate a minimal document, let's put some data in the report. To do this, we will make the following changes:
Ahora que nuestro programa puede generar un documento mínimo, introduzcamos algunos datos en el informe. Para ello, haremos los siguientes cambios:

```
#!/bin/bash
# Program to output a system information page
echo "<html>
    <head>
        <title>System Information Report</title>
    </head>
    <body>
        <h1>System Information Report</h1>
    </body>
</html>"
```

We added a page title and a heading to the body of the report.
Agregamos un título de página y un encabezado al cuerpo del informe.

Variables and Constants
Variables y constantes

There is an issue with our script, however. Notice how the string System Information Report is repeated? With our tiny script it's not a problem, but let's imagine that our script was really long and we had multiple instances of this string. If we wanted to change the title to something else, we would have to change it in multiple places, which could be a lot of work. What if we could arrange the script so that the string appeared only once and not multiple times? That would make future maintenance of the script much easier. Here's how we could do that:
Sin embargo, hay un problema con nuestro script. ¿Observa cómo se repite la cadena Informe de información del sistema? Con nuestro pequeño script no es un problema, pero imaginemos que nuestro script era muy largo y teníamos varias instancias de esta cadena. Si quisiéramos cambiar el título por otro, tendríamos que cambiarlo en varios lugares, lo que podría suponer mucho trabajo. ¿Qué pasaría si pudiéramos organizar el guión para que la cadena apareciera solo una vez y no varias veces? Eso facilitaría mucho el mantenimiento futuro del guión. Así es como podemos hacer eso:

```
#!/bin/bash
# Program to output a system information page

title="System Information Report"

echo "<html>
        <head>
            <title>$title</title>
        </head>
        <body>
            <h1>$title</h1>
        </body>
</html>"
```

By creating a variable named title and assigning it the value System Information Report , we can take advantage of parameter expansion and place the string in multiple locations.
Al crear una variable llamada título y asignarle el valor Informe de información del sistema, podemos aprovechar la expansión de parámetros y colocar la cadena en varias ubicaciones.

So, how do we create a variable? Simple, we just use it. When the shell encounters a variable, it automatically creates it. This differs from many programming languages in which variables must be explicitly declared or defined before use. The shell is very lax about this, which can lead to some problems.
Entonces, ¿cómo creamos una variable? Simple, solo lo usamos. Cuando el shell encuentra una variable, la

crea automáticamente. Esto difiere de muchos lenguajes de programación en los que las variables deben declararse o definirse explícitamente antes de su uso. El caparazón es muy laxo al respecto, lo que puede ocasionar algunos problemas.

For example, consider this scenario played out on the command line:
Por ejemplo, considere este escenario desarrollado en la línea de comando:

```
[~]$ foo="yes"
[~]$ echo $foo
yes
[~]$ echo $fool
[~]$
```

We first assign the value yes to the variable foo , and then we display its value with echo . Next we display the value of the variable name misspelled as fool and get a blank result. This is because the shell happily created the variable fool when it encountered it and gave it the default value of nothing, or empty. From this, we learn that we must pay close attention to our spelling! It's also important to understand what really happened in this example. From our previous look at how the shell performs expansions, we know that the following command:
Primero asignamos el valor yes a la variable foo, y luego mostramos su valor con echo. A continuación, mostramos el valor del nombre de la variable mal escrito como tonto y obtenemos un resultado en blanco. Esto se debe a que el shell creó felizmente la variable tonto cuando la encontró y le dio el valor predeterminado de nada o vacío. De esto, aprendemos que debemos prestar mucha atención a nuestra ortografía. También es importante comprender lo que realmente sucedió en este ejemplo. De nuestra mirada anterior a cómo el shell realiza expansiones, sabemos que el siguiente comando:

```
[~]$ echo $foo
```

undergoes parameter expansion and results in the following:
sufre una expansión de parámetros y da como resultado lo siguiente:

```
[~]$ echo yes
```

By contrast, the following command:
Por el contrario, el siguiente comando:

```
[~]$ echo $fool
```

expands into this:
se expande en esto:

```
[~]$ echo
```

The empty variable expands into nothing! This can play havoc with commands that require arguments. Here's an example:
¡La variable vacía se expande hasta convertirse en nada! Esto puede causar estragos en los comandos que requieren argumentos. Aquí tienes un ejemplo:

```
[~]$ foo=foo.txt
[~]$ foo1=foo1.txt
[~]$ cp $foo $fool
cp: missing destination file operand after `foo.txt'
Try `cp --help' for more information.
```

We assign values to two variables, foo and foo1 . We then perform a cp but misspell the name of the second argument. After expansion, the cp command is sent only one argument, though it requires two.
Asignamos valores a dos variables, foo y foo1. Luego realizamos un cp pero escribimos mal el nombre del segundo argumento. Después de la expansión, el comando cp se envía solo un argumento, aunque requiere dos.

There are some rules about variable names:
Existen algunas reglas sobre los nombres de variables:

> Variable names may consist of alphanumeric characters (letters and numbers) and underscore characters.
> Los nombres de las variables pueden consistir en caracteres alfanuméricos (letras y números) y caracteres de subrayado.

> The first character of a variable name must be either a letter or an underscore.
> El primer carácter de un nombre de variable debe ser una letra o un guión bajo.

> Spaces and punctuation symbols are not allowed.
> No se permiten espacios ni símbolos de puntuación.

The word variable implies a value that changes, and in many applications, variables are used this way. However, the variable in our application, title, is used as a constant. A constant is just like a variable in that it has a name and contains a value. The difference is that the value of a constant does not change. In an application that performs geometric calculations, we might define PI as a constant and assign it the value of 3.1415, instead of using the number literally throughout our program. The shell makes no distinction

between variables and constants; they are mostly for the programmer's convenience. A common convention is to use uppercase letters to designate constants and lowercase letters for true variables. We will modify our script to comply with this convention:
La palabra variable implica un valor que cambia y, en muchas aplicaciones, las variables se utilizan de esta manera. Sin embargo, la variable de nuestra aplicación, title, se usa como constante. Una constante es como una variable en el sentido de que tiene un nombre y contiene un valor. La diferencia es que el valor de una constante no cambia. En una aplicación que realiza cálculos geométricos, podríamos definir PI como una constante y asignarle el valor de 3,1415, en lugar de usar el número literalmente en todo nuestro programa. El shell no hace distinciones entre variables y constantes; son principalmente para la conveniencia del programador. Una convención común es usar letras mayúsculas para designar constantes y letras minúsculas para variables verdaderas. Modificaremos nuestro script para cumplir con esta convención:

```
#!/bin/bash
# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"

echo "<html>
    <head>
        <title>$TITLE</title>
    </head>
    <body>
        <h1>$TITLE</h1>
    </body>
</html>"
```

We also took the opportunity to jazz up our title by adding the value of the shell variable HOSTNAME . This is the network name of the machine.
También aprovechamos la oportunidad para mejorar nuestro título agregando el valor de la variable de shell HOSTNAME. Este es el nombre de red de la máquina.

> Note
> Nota

> The shell actually does provide a way to enforce the immutability of constants, through the use of the declare built-in command with the -r (read-only) option.
> El shell realmente proporciona una forma de imponer la inmutabilidad de las constantes, mediante el uso del comando declare incorporado con la opción -r (solo lectura).

> Had we assigned TITLE this way:
> Si hubiéramos asignado TITLE de esta manera:
>
> ```
> declare -r TITLE="Page Title"
> ```

> the shell would prevent any subsequent assignment to TITLE . This feature is rarely used, but it exists for very formal scripts.
> el shell evitaría cualquier asignación posterior a TITLE. Esta función se utiliza con poca frecuencia, pero existe para scripts muy formales.

Assigning Values to Variables and Constants
Asignar valores a variables y constantes

Here is where our knowledge of expansion really starts to pay off. As we have seen, variables are assigned values this way:
Aquí es donde nuestro conocimiento de la expansión realmente comienza a dar sus frutos. Como hemos visto, a las variables se les asignan valores de esta manera:

```
variable=value
```

where variable is the name of the variable and value is a string. Unlike some other programming languages, the shell does not care about the type of data assigned to a variable; it treats them all as strings. You can force the shell to restrict the assignment to integers by using the declare command with the -i option, but, like setting variables as read-only, this is rarely done.
donde variable es el nombre de la variable y valor es una cadena. A diferencia de otros lenguajes de programación, al shell no le importa el tipo de datos asignados a una variable; los trata a todos como cadenas. Puede forzar al shell a restringir la asignación a números enteros usando el comando declare con la opción -i, pero, al igual que configurar las variables como de solo lectura, esto rara vez se hace.

Note that in an assignment, there must be no spaces between the variable name, the equal sign, and the value. So, what can the value consist of? It can have anything that we can expand into a string.
Tenga en cuenta que en una asignación, no debe haber espacios entre el nombre de la variable, el signo igual y el valor. Entonces, ¿en qué puede consistir el valor? Puede tener cualquier cosa que podamos expandir en una cadena.

```
a=z # Assign the string "z" to variable a.
b="a string" # Embedded spaces must be within quotes.
c="a string and $b" # Other expansions such as variables can be
        # expanded into the assignment.
d="$(ls -l foo.txt)" # Results of a command.
e=$((5 * 7)) # Arithmetic expansion.
f="\t\ta string\n" # Escape sequences such as tabs and newlines.
```

Multiple variable assignments may be done on a single line.
Se pueden realizar múltiples asignaciones de variables en una sola línea.

```
a=5 b="a string"
```

During expansion, variable names may be surrounded by optional braces, {} . This is useful in cases where a variable name becomes ambiguous because of its surrounding context. Here, we try to change the name of a file from myfile to myfile1, using a variable:
Durante la expansión, los nombres de las variables pueden estar rodeados por llaves opcionales, {}. Esto es útil en los casos en que el nombre de una variable se vuelve ambiguo debido a su contexto circundante. Aquí, intentamos cambiar el nombre de un archivo de myfile a myfile1, usando una variable:

```
[~]$ filename="myfile"
[~]$ touch "$filename"
[~]$ mv "$filename" "$filename1"
mv: missing destination file operand after `myfile'
Try `mv --help' for more information.
```

This attempt fails because the shell interprets the second argument of the mv command as a new (and empty) variable. The problem can be overcome this way:
Este intento falla porque el shell interpreta el segundo argumento del comando mv como una variable nueva (y vacía). El problema se puede solucionar de esta manera:

```
[~]$ mv "$filename" "${filename}1"
```

By adding the surrounding braces, the shell no longer interprets the trailing 1 as part of the variable name.
Al agregar las llaves circundantes, el shell ya no interpreta el 1 final como parte del nombre de la variable.

> Note
> Nota

> It's good practice is to enclose variables and command substitutions in double quotes to limit the effects of word-splitting by the shell. Quoting is especially important when a variable might contain a filename.
> Es una buena práctica encerrar las variables y ordenar las sustituciones entre comillas dobles para limitar los efectos de la división de palabras por el shell. Las comillas son especialmente importantes cuando una variable puede contener un nombre de archivo

We'll take this opportunity to add some data to our report: namely, the date and time the report was created and the username of the creator.
Aprovecharemos esta oportunidad para agregar algunos datos a nuestro informe: a saber, la fecha y la hora en que se creó el informe y el nombre de usuario del creador.

```
#!/bin/bash
# Program to output a system information page
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME="$(date +"%x %r %Z")"
TIMESTAMP="Generated $CURRENT_TIME, by $USER"
echo "<html>
    <head>
        <title>$TITLE</title>
    </head>
    <body>
        <h1>$TITLE</h1>
        <p>$TIMESTAMP</p>
    </body>
</html>"
```

Here Documents
Aquí Documentos

We've looked at two different methods of outputting our text, both using the echo command. There is a third way called a here document or here script. A here document is an additional form of I/O redirection in which we embed a body of text into our script and feed it into the standard input of a command. It works like this:
Hemos analizado dos métodos diferentes para generar nuestro texto, ambos usando el comando echo. Hay una tercera forma llamada documento here o script here. Un documento aquí es una forma adicional de redirección de E / S en la que incrustamos un cuerpo de texto en nuestro script y lo introducimos en la entrada estándar de un comando. Funciona así:

```
command << token
text
token
```

where command is the name of command that accepts standard input and token is a string used to indicate the end of the embedded text. Here we'll modify our script to use a here document:
donde comando es el nombre del comando que acepta entrada estándar y token es una cadena que se usa para indicar el final del texto incrustado. Aquí modificaremos nuestro script para usar un documento aquí:

```
#!/bin/bash

# Program to output a system information page
```

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

cat << _EOF_
<html>
    <head>
        <title>$TITLE</title>
    </head>
    <body>
        <h1>$TITLE</h1>
        <p>$TIMESTAMP</p>
    </body>
</html>
_EOF_
```

Instead of using echo , our script now uses cat and a here document. The string *EOF* (meaning end of file, a common convention) was selected as the token and marks the end of the embedded text. Note that the token must appear alone and that there must not be trailing spaces on the line.
En lugar de usar echo, nuestro script ahora usa cat y un documento here. La cadena *EOF* (que significa final de archivo, una convención común) se seleccionó como el símbolo y marca el final del texto incrustado. Tenga en cuenta que el token debe aparecer solo y que no debe haber espacios al final de la línea.

So, what's the advantage of using a here document? It's mostly the same as echo , except that, by default, single and double quotes within here documents lose their special meaning to the shell. Here is a command line example:
Entonces, ¿cuál es la ventaja de utilizar un documento aquí? Es casi lo mismo que echo, excepto que, de forma predeterminada, las comillas simples y dobles dentro de estos documentos pierden su significado especial para el shell. Aquí hay un ejemplo de línea de comando:

```
[~]$ foo="some text"
[~]$ cat << _EOF_
> $foo
> "$foo"
> '$foo'
> \$foo
> _EOF_
some text
"some text"
'some text'
$foo
```

As we can see, the shell pays no attention to the quotation marks. It treats them as ordinary characters. This allows us to embed quotes freely within a here document. This could turn out to be handy for our report program.

Here documents can be used with any command that accepts standard input. In this example, we use a here document to pass a series of commands to the ftp program to retrieve a file from a remote FTP server:
Como podemos ver, el caparazón no presta atención a las comillas. Los trata como personajes ordinarios. Esto nos permite insertar citas libremente dentro de un documento aquí. Esto podría resultar útil para nuestro programa de informes.

```
#!/bin/bash

# Script to retrieve a file via FTP

FTP_SERVER=ftp.nl.debian.org FTP_PATH=/debian/dists/stretch/main/installer-
amd64/current/images/cdrom REMOTE_FILE=debian-cd_info.tar.gz

ftp -n << _EOF_
open $FTP_SERVER
user anonymous me@linuxbox
cd $FTP_PATH
hash
get $REMOTE_FILE
bye
_EOF_
ls -l "$REMOTE_FILE"
```

If we change the redirection operator from << to <<- , the shell will ignore leading tab characters (but not spaces) in the here document. This allows a here document to be indented, which can improve readability.
Si cambiamos el operador de redirección de << a << -, el shell ignorará los caracteres de tabulación iniciales (pero no los espacios) en el documento here. Esto permite sangrar un documento aquí, lo que puede mejorar la legibilidad.

```
#!/bin/bash

# Script to retrieve a file via FTP

FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/stretch/main/installer-amd64/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n <<- _EOF_
    open $FTP_SERVER
    user anonymous me@linuxbox
    cd $FTP_PATH
    hash
    get $REMOTE_FILE
    bye
    _EOF_
ls -l "$REMOTE_FILE"
```

This feature can be somewhat problematic because many text editors (and programmers themselves) will prefer to use spaces instead of tabs to achieve indentation in their scripts.
Esta característica puede ser algo problemática porque muchos editores de texto (y los mismos programadores) preferirán usar espacios en lugar de pestañas para lograr sangrías en sus scripts.

Summing Up
Resumen

In this chapter, we started a project that will carry us through the process of building a successful script. We introduced the concept of variables and constants and how they can be employed. They are the first of many applications we will find for parameter expansion. We also looked at how to produce output from our script and various methods for embedding blocks of text.
En este capítulo, comenzamos un proyecto que nos llevará a través del proceso de construcción de un guión exitoso. Introdujimos el concepto de variables y constantes y cómo se pueden emplear. Son la primera de muchas aplicaciones que encontraremos para la expansión de parámetros. También analizamos cómo producir resultados a partir de nuestro script y varios métodos para incrustar bloques de texto.