25

STARTING A PROJECT

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.

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.

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. That will come later.

First Stage: Minimal Document

The first thing we need to know is the format of a well-formed HTML document. It looks like this:

```
<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*.

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*.

```
[me@linuxbox ~]$ vim ~/bin/sys_info_page
```

Enter the following program:

```
#!/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>"
```

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. After saving the file, we'll make it executable and attempt to run it.

```
[me@linuxbox ~]$ chmod 755 ~/bin/sys_info_page
[me@linuxbox ~]$ 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.

```
[me@linuxbox ~]$ sys_info_page > sys_info_page.html
[me@linuxbox ~]$ firefox sys_info_page.html
```

So far, so good.

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:

```
#!/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:

```
[me@linuxbox ~]$ 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. This feature is a little obscure right now, but later, when we cover multiline programming statements, it will turn out to be quite handy.

Second Stage: Adding a Little Data

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:

```
#!/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.

Variables and Constants

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:

```
#!/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.

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. For example, consider this scenario played out on the command line:

```
[me@linuxbox ~]$ foo="yes"
[me@linuxbox ~]$ echo $foo
yes
[me@linuxbox ~]$ echo $fool
[me@linuxbox ~]$
```

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:

```
[me@linuxbox ~]$ echo $foo
```

undergoes parameter expansion and results in the following:

```
[me@linuxbox ~]$ echo yes
```

By contrast, the following command:

```
[me@linuxbox ~]$ echo $fool
```

expands into this:

```
[me@linuxbox ~]$ echo
```

The empty variable expands into nothing! This can play havoc with commands that require arguments. Here's an example:

```
[me@linuxbox ~]$ foo=foo.txt
[me@linuxbox ~]$ foo1=foo1.txt
[me@linuxbox ~]$ 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.

There are some rules about variable names:

- Variable names may consist of alphanumeric characters (letters and numbers) and underscore characters.
- The first character of a variable name must be either a letter or an underscore.
- Spaces and punctuation symbols are not allowed.

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:

```
#!/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.

> **NOTE**
>
> *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. Had we assigned `TITLE` this way:*

```
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.

Assigning Values to Variables and Constants

Here is where our knowledge of expansion really starts to pay off. As we have seen, variables are assigned values this way:

```
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.

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.

```
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.

```
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:

```
[me@linuxbox ~]$ filename="myfile"
[me@linuxbox ~]$ touch "$filename"
[me@linuxbox ~]$ 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:

```
[me@linuxbox ~]$ mv "$filename" "${filename}1"
```

By adding the surrounding braces, the shell no longer interprets the trailing `1` as part of the variable name.

NOTE

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.

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.

```
#!/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

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:

```
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:

```
#!/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.

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:

```
[me@linuxbox ~]$ foo="some text"
[me@linuxbox ~]$ 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:

```
#!/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.

```
#!/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.

Summing Up

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.

Support Sign Out

©2022 O'REILLY MEDIA, INC. TERMS OF SERVICE PRIVACY POLICY