Here Docs

In this lesson, we will cover 'here docs' and 'here strings', and when to use them.

WE'LL COVER THE FOLLOWING

> Note: In this lesson, leading spaces are tabs. We have covered this in a previous lesson, but as a reminder: to get a tab character in your shell, type `\C-v`, and then hit the `TAB` button.

How Important is this Lesson?

Here documents, once learned, are used frequently in bash scripts or on the command line. It's a piece of knowledge that separates the experienced user from the junior.

Basic Here Docs

Type this in to see the basic form of the here doc:

The first line starts with cat followed by a redirection to a file.

Then you use two left chevrons and follow that with a string that represents a marker for the end of the file's contents. In this case, the word is `END`.

```
cat > afile.txt << END
```

Type the above code into the terminal in this lesson.

Then you type in whatever you want the file to contain.

```
    A file can contain
            whatever you like
```

Type the above code into the terminal in this lesson.

(The whitespace in the second line above are spaces.)

When you're done, you can finish the file by typing the string you used as a marker on the first line alone on its own line:

```
END
```

Type the above code into the terminal in this lesson.

You can check the contents of the file by just `cat`ing it:

```
cat afile.txt
```

Type the above code into the terminal in this lesson.

The marker word does not need to be `END`! It could be anything you choose. `END` is generally used as a convention. Sometimes you see `EOF`, or `STOP`, or something similar. If you have a document with `END` in it, for example, you might want to avoid problems with the document ending early by choosing a different word.

More Advanced Here Docs

Now you're going to put a *here document* in a function. The function takes one argument. This argument is used as a filename, and the function creates a simple script with that filename that echoes the first argument given to that script.

Will this work? Read it carefully, predict the outcome, and then run it:

```
$ function write_echoer_file {
    cat > $1 << END
#!/bin/bash
echo $1
END
    chmod +x $1
}
write_echoer_file echoer
./echoer 'Hello world'
```

Type the above code into the terminal in this lesson.

- **Lines 1-7** creates a function called `write_echoer`
- **Lines 2-5** creates a file with the name of the first argument to the `write_echoer_file` function that `echo`es its own name
- **Line 6** makes the newly-created file executable
- **Line 8** uses `write_echoer_file` to create a program called `echoer`
- **Line 9** tries to call the newly-created `echoer` program to output `Hello world`

Hmmm. That didn't work, because the `$1` got interpreted in the write_echoer_file function as being the filename we passed in. In the *here doc*, we wanted the `$1` characters to be put into the script without being interpreted.

Try this instead:

```
function write_echoer_file {
    cat > $1 << 'END'
#!/bin/bash
echo $1
END
    chmod +x $1
}
write_echoer_file echoer
./echoer 'Hello world'
```

Type the above code into the terminal in this lesson.

Do you see the difference? This time, the *delimiter* word `END` was wrapped in single quotes. This made sure that the `echo $1` was not interpreted by the

shell when being typed in.

Can you see why we needed to use single quotes here? What happens when you use double quotes?

This kind of confusion can happen all the time when writing bash scripts, so it's really important to get these differences clear in your mind.

Our function is working now, but we could still make it better.

Try this (remember, the leading spaces are tabs - see the note above for how to input a tab):

```
function write_echoer_file {
    cat > $1 <<- 'END'
        #!/bin/bash
        echo $1
        END
    chmod +x $1
}
write_echoer_file echoer
./echoer
```

Type the above code into the terminal in this lesson.

The difference there was that we added a hyphen after the `<<` in **line 2**. This tells the shell to ignore leading tabs in the following text. This makes it much neater and easier to read scripts that use here docs, as the indentation can be consistent.

What if `END` is part of the *here doc* within another line?

```
function write_echoer_file {
    cat > $1 <<- 'END'
        #!/bin/bash
        echo $1
        echo Is this the END?
        END
    chmod +x $1
}
write_echoer_file echoer
./echoer
```

Type the above code into the terminal in this lesson.

No problem if it is not the only thing on the line.

Experiment to try and see what happens if it is.

Here Strings

Related to the *here doc*, a *here string* can be applied in the same way with the `<<<` operator:

```
function write_here_string_to_file {
    cat > $1 <<< $2
}
write_here_string_to_file afile.txt "Write this out"
```

Type the above code into the terminal in this lesson.

Here Docs Quiz

Q

What will this do?

```
HOME=asd
cat > afile << END
my home is $HOME
END
```

What You Learned

- What *here documents* are
- What *here strings* are
- How to create a *here document*
- How *here docs* and variables can be appropriately handled
- How to use *here docs* in a way that looks neat in a shell script

What Next?

Next we look at how bash maintains and uses a history of the commands run within it.

Exercises

1) Try passing a multi-line string to a here string. What happens?