

Linux Guide

Aaron Baw

January 10, 2017

Contents

1	I/O Streams	1
1.1	I/O Uses in UNIX	1
1.2	I/O Redirection	2
2	Linux Filesystem Permissions	3
2.1	Understanding the permissions system on linux	3
2.2	Changing permissions on linux	4
2.2.1	Using numerical values to assign permissions	4
3	Bash Scripting	5
3.1	Declaring variables in Bash	5
3.2	Whitespace in Bash	5
3.3	IF Statements in Bash	6
3.4	FOR Loops in Bash	7
3.5	Arrays in Bash	7
3.6	Functions in Bash	8
3.7	Variable scope in Bash	8

1 I/O Streams

Standard streams are a standardized way that different pieces of software and devices can communicate with each other. It serves as a protocol that developers can use to move data from outputs of software to inputs of other software, in a kind of agnostic manner.

As per the convention there are 3 standard I/O streams, namely **Standard Input**(stdin), **Standard Output**(stdout) and **Standard Error**(stderr).[1]

1.1 I/O Uses in UNIX

I/O Streams play a fundamental part of the inter-workings of UNIX, particularly when it comes to the use of the shell. As per the UNIX 'conise-and-modular'

philosophy, having an abstracted means of communication between pieces of software forms a crucial role in the interoperability of the operating system.

The result of this system is the ability to connect different pieces of software together to achieve powerful functionality.

By default, when using the shell, the **stdout** stream tends to be the text output of the shell, whereas the **stdin** stream tends to be the input from the keyboard, terminated by the enter key on most occasions.

1.2 I/O Redirection

One of the many advantages of using Standard Streams is the ability to manipulate the data destinations and origins using some of the in-built tools and commands provided by UNIX.

An example of this would be through the use of pipe `|`, a powerful redirection command.

The pipe command allows the stdout of one command to be redirected to the stdin of another command, instead of the default text output stdout of the terminal, for instance.

```
cat textDocument.txt | grep 'john'
```

Figure 1: An example of stdin/stdout redirection with the pipe command.

The greater than symbol `>` can also be used to redirect the stdout of a command to write a file rather than display on the terminal output.

```
echo "A message" > message.txt
```

Figure 2: An example of stdin/stdout redirection with the `>` command.

Below is a summary of some of the useful commands that can be used to redirect Standard Streams:

- `|`: Pipe. Used to redirect stdout of one command into the stdin of another.
- `>`: Used to redirect stdout of a command to be written to a specified file.
- `>>`: Same as `>` but instead appends to the beginning of the file if a file already exists.
- `<`: Used to read a file and send this as the stdin of a command.

These commands are very versatile, and are able to operate with each other to achieve a wide scope of functionality. Furthermore, they can also be nested within each other.

In the example above a text file is being read, and then piped into the **sort** command, of which the output of this command is then piped into **sed** where the first row is being selected, and finally this output is then sent to **column**

```
cat data.txt | sort -n | sed -n 1,1p | column -t >
processedData.txt
```

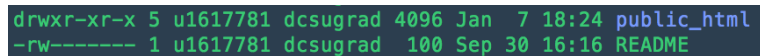
Figure 3: Combining and nesting I/O redirection commands.

which organizes the output into a column. After the data has been processed, the `>` command writes the final output from the **column** command to a file ‘**processedData.txt**’.

2 Linux Filesystem Permissions

2.1 Understanding the permissions system on linux

The permissions for a given file or directory in linux can be viewed with the **ls -l** command.



```
drwxr-xr-x 5 u1617781 dcsugrad 4096 Jan  7 18:24 public_html
-rw----- 1 u1617781 dcsugrad 100 Sep 30 16:16 README
```

Figure 4: Sample output for the **ls -l** command illustrating linux file permissions.

[2]

As you can see above, the permissions for each file is listed in the left-most column of the command output. There are four key components to the permissions layout:

- Each portion of the permissions string represents a different feature. The first character states if the item is a file (denoted by `-`), a directory (denoted by `d`), or a symbolic link (denoted by `l`)
- The rest of the permissions string states the level of access each entity on the system has to that particular file. Out of the next 9 characters, the first 3 represent the permissions for the owner of the file (typically the creator), then the next 3 characters represent the permissions for the user group, and finally the last 3 characters represent the permissions for all users.
- Each subsection of 3 characters as outlined above use the same system to illustrate the level of permissions a particular system entity has for the file. The presence of **r** means that the entity can read the file, **w** means that the entity can write to the file, and finally **x** means that the entity can execute the file. Where a `-` character is used in place of what would otherwise be a **r**, **w** or **x**, it means that this particular permission is omitted from the entity.

2.2 Changing permissions on linux

A popular way of altering the permissions on the linux filesystem is to use the **chmod** command.

There are a number of different ways that this can be used. Firstly, the perhaps clearest way to alter commands of a particular file is through the following general syntax:

```
chmod a+x filename
```

Figure 5: Simple alteration of filesystem permissions on linux with the **chmod** command.

Of the first argument passed to the command (**a+x**), where the **'a'** character is positioned stands for the entity that is being targeted. The **'a'** character stands for 'all users'. Other options include **'o'** (file owner) and **'g'** (user group).

Where the **'+'** lies indicates whether the following specified permissions should be added (indicated by the **+** in this instance), or removed, (which would be denoted by the presence of a **-** symbol), from the preceeding system entity.

Finally, the last symbol position, in this example the **'x'** character, indicates what permissions should be altered (in this case added). In this instance, the **x** character stands for 'eXecute' as may be implied. Other options include **r** (read) and **w** (write).

Overall, the semantic meaning of the example chmod usage above would be to "add execute permissions for all users on the system."

The equation below summarizes the functionality of this particular usage of the **chmod** command:

$$chmod \quad a/o/g \pm r/w/x \quad filename \quad (1)$$

2.2.1 Using numerical values to assign permissions

In linux, filesystem permissions can also be represented by numerical values. For example, a permissions value of **755** would mean 'rwxr-xr-x' in permissions-string form. Much like earlier, each numerical position represents the permissions that each corresponding entity would have; in the same owner-usergroup-allusers order that was outlined earlier.

Each permission **r**, **w** or **x** has a specific numerical value as can be seen in figure 6.[3]

The individual values of each of the permissions can be added together to form the permissions for each system entity. For example, if you were to only desire read and execute capability for a particular file, you would only need to add up the corresponding values, in this case being **5**. If you desired all other

- **r**: 4
- **w**: 2
- **x**: 1

Figure 6: Permission numerical values.

entities to have the same permissions, then the total permission value would be **555**.

Finally, this permission value can be applied to any file just as one would expect with the **chmod** command.

```
chmod 555 filename
```

Figure 7: Assigning permissions directly with chmod.

3 Bash Scripting

Bash scripting forms a major part of the linux operating system experience. The interoperability of modular commands and standard streams means that combining commands to produce new functionality becomes quite intuitive.

3.1 Declaring variables in Bash

Variables in bash are loosely-typed meaning that variable types are not specified. Below is an example of a variable declaration in bash:

```
text="A message goes here"
sentence="$text and is used here"
```

Figure 8: An example of variable declaration and usage.

As you can see above, variables are used with the **\$** prefix before the variable name. This is a requirement anytime the value of a variable is desired to be used.

3.2 Whitespace in Bash

Something important to note about programming in bash is the significance of **whitespace**. With the exception of a few programming languages like **Python**, for example, most programming languages are fairly agnostic about whitespace. This is not the case in bash.

When declaring variables for instance, there must not be any whitespace between the name of the variable, the equals (=) sign, and the variable value itself.

Another prevalent example of the importance of keeping whitespace in mind is with the usage of **IF** statements and **FOR** loops, for example.

When declaring an **IF** statement, it is crucial that when using the `[[]]` (double bracket) evaluation, that exactly **one** space is present between each portion of the comparison.

When declaring **IF** statements for example, succeeding the comparison requires a newline and the word ‘**then**’, after which another newline is inserted and the conditional logic of the **IF** statement can be applied. Without the presence of these newlines, semicolons are required to tell the interpreter where a newline would otherwise be.

3.3 IF Statements in Bash

As mentioned above, whitespace is something important to keep in mind when constructing IF statements in Bash.

Unlike other programming languages which typically evaluate the truthfulness of a statement, in Bash all that is conditional to run a following piece of code is the **exit status** of the statement. With a successful exit status, the conditional code block will then execute, otherwise it will not.

To use IF statements in a more traditional manner, it is possible to utilize `[]` syntax after the **IF** declaration. The code placed within the brackets (keeping in mind the significance of whitespace as outlined previously) is evaluated much like if it were forming part of the arguments to the **test** command.

As such it is possible to run code if a certain value is greater than another value, for example:

```
if [ "$var1" -gt "$var2" ]
then
    # Do something
fi
```

Figure 9: Declaring an IF Statement in Bash

There is also a more traditional syntax that can be used to declare the conditions for IF statements within bash using a double square bracket (`[[]]`) as opposed to just a single one.

It is still important to note the necessity of a single whitespace between each portion of the statement, particularly between the variable names and the brackets, as well as in between everything else.

```

if [[ $var1 > $var2 ]]
then
    # Do something
fi

```

Figure 10: Declaring an IF Statement in Bash with more familiar syntax.

3.4 FOR Loops in Bash

FOR loops in bash behave quite similarly to that of other familiar programming languages. It is interesting to note that there are some quirky features of bash that allow for some interesting functionality.

If, for example, it would be desired to iterate through names from a particular text document, an easy way to do this in Bash would be the following:

```

for a in $names
do
    echo $a # Print out every name for e.g.
done

```

Figure 11: A useful but simple use of **FOR** in Bash.

If it is desirable to loop over a specific set of numbers, then **seq** can be used, or a more traditional syntax:

```

for (( i=0;i<$limit;i++))
do
    # Do something.
done

```

Figure 12: A more traditional for loop syntax in Bash.

FOR loops are particularly useful when used in conjunction with **arrays** as covered in next chapter.

3.5 Arrays in Bash

Arrays operate much in the same way as one would expect in other programming languages. Arrays in Bash can be declared by treating a variable as an array; that is targeting an element with an index such as for example **array[\$PositiveInteger]** which would turn the variable ‘\$array’ into an array. Alternatively, arrays can also be declared using the **declare -a** command.

Once arrays have been declared and filled, each element can be accessed with a FOR loop. It is possible to use the **for a in \$variablename...** syntax for

```
declare -a nameOfArray
```

Figure 13: One way to declare an array in Bash.

this, or if index numbers are required, the following syntax can be used:

```
for (( i=0; i<$(( $array )); i++ ))  
do  
    if [[ $i % 2 == 0 ]]  
    then  
        echo ${array[ $i ]}  
    fi  
done
```

Figure 14: Looping through an indexed array in Bash.

3.6 Functions in Bash

To declare functions in bash, there are a few different types of syntax that can be used, such as:

```
function printVariable {  
    echo ‘‘Argument passed: $1’’  
}
```

Figure 15: Declaring a function in Bash.

Arguments are passed to functions much like arguments are passed to scripts and Bash programs in general, with the syntax **functionName \$Argument1 \$Argument2 ...** and so on.

Functions do not return values much like in other programming languages, but rather exit codes much like the Bash program as a whole. If it is required to use the declared function as part of a certain block of code to for example perform some logic, the output of this can be assigned to a variable as follows:

The use of the dollar sign and parenthesis **\$(...)** tells the interpreter to save any stdout of the function to the variable.

3.7 Variable scope in Bash

When it comes to defining functions to be used in Bash, something to note is the **scope** of each of the variables.

Each variable that is declared **within** a function, can **only** be used within that function. That is, its scope is only that of its nearest parent code block.


```
variable=$(calculate $argument)
```

Figure 16: Saving the result of a function.

Variables that are declared outside of the scope of any function or code block such as an IF statement or FOR loop have a **global scope** and can be used within any child code blocks.

The same is true for any variable declaration; child code blocks inherit the variable scope of their parents.

References

- [1] “nixCraft: redirecting io.” <https://www.cyberciti.biz/faq/redirecting-stderr-to-stdout/>. Accessed: 2017-01-04.
- [2] “Permissions screenshot.” Taken from MacOS Terminal on 2017-01-02.
- [3] “Permissions calculator.” <http://permissions-calculator.org/>. Accessed: 2017-01-05.