# Pipes and Redirects

In this lesson you'll look at basic redirection and pipes. You'll also learn about file descriptors, 'standard out' and 'standard error' and 'special' files like '/dev/null'.

WE'LL COVER THE FOLLOWING

**Pipes and redirects** are used very frequently in bash and by all levels of user.

This can cause a problem. They are used so often by all users of bash that many don't understand their subtleties, how they work, or their full power.

## How Important is this Lesson?

This lesson is essential.

## Basic Redirects

Start off by creating a file:

```
echo "contents of file1" > file1
```

Type the above code into the terminal in this lesson.

Terminal

> Note: continue to use this terminal during this lesson. Throughout this course it is assumed that you complete each lesson by typing in the commands in the provided terminal for that lesson in the order the commands are given.

The `>` character is the *redirect* operator. This takes the output from the preceding command that you'd normally see in the terminal and sends it to a file that you give it. Here it creates a file called `file1` and puts the `echo`ed string into it. Try `cat`ing the file if you want to check by running:

```
cat file1
```

There's a subtlety here which we'll get to: sometimes not all the output you see in the terminal would get redirected by this, but don't worry about this yet.

## Basic Pipes

Now type this in:

```
cat file1 | grep -c file
```

Type the above code into the terminal in this lesson.

> Note: If you don't know what `grep` is, you will need to learn. This is a good place to start: https://en.wikipedia.org/wiki/Grep

Normally you'd run a `grep` with the filename as the last argument, but instead here we **pipe** the contents of `file1` into the `grep` command by using the 'pipe' operator: `|`. The resulting output of `1` indicates the number lines that matched the word `file` in the the file called `file1`.

A *pipe* takes the standard output of one command and passes it as the input to another. What, then is **standard output**, really? You will find out soon!

```
cat file2
```

Type the above code into the terminal in this lesson.

You should have seen an error, because the file did not exist.

Now run this, and try and guess the result before you run it:

```
cat file2 | grep -c file
```

Type the above code into the terminal in this lesson.

You should have seen two lines. The first will be an error

```
cat: file2: No such file or directory
```

and the second line a `0`.

We will explain further below why a `0` is shown, even though a line was outputted that had the word `file` in it.

> Note: See Exercise 3 below for an often-useful operator that changes this behaviour.

## Standard Output vs Standard Error #

In addition to **standard output**, there is also a **standard error** channel. When you pass a non-existent file to the `cat` command, it throws an error message out to the terminal, as you would have seen when you attempted the code above. Although the message looks the same as the contents of a file, it is sent to a different output channel. In this case it’s *standard error* rather than *standard output*.

As a result, it is NOT passed through the pipe to the `grep` command, and `grep` counts zero matches in its output.

To the viewer of the terminal, there is no difference, but to bash there is all the difference in the world!

There is a simpler way to refer to these channels. A number is assigned to each of them by the operating system.

These are the numbered *file descriptors*, and the first three are assigned to the numbers `0`, `1` and `2`.

- `0` is *standard input*
- `1` is *standard output*
- `2` is *standard error*

When you redirect 'standard output' to a file, you use the redirection operator `>`. Implicitly, you are using the `1` file descriptor.

Type this to see an example of redirecting `2`, which is 'standard error'.

```
command_does_not_exist
command_does_not_exist 2> /dev/null
```

Type the above code into the terminal in this lesson.

- **Line 1** above will show an error as you might expect, but **line 2** does not
- In **line 2**, the file descriptor `2` (standard error) is directed to a file called `/dev/null`

The file `/dev/null` is a special file created by Linux (and UNIX) kernels. It is effectively a black hole into which data can be pumped: anything written to it will be absorbed and ignored.

Another commonly seen redirection operator is `2>&1`:

```
command_does_not_exist 2>&1
```

Type the above code into the terminal in this lesson.

What this does is tell the shell to send the output on standard error (`2`) to whatever endpoint standard output is pointed to at that point in the command.

Since standard output is pointed at the terminal at that time, standard error is also pointed at the terminal. From your point of view you see no difference, since both standard output and standard error are pointed at the terminal anyway.

But when we try and redirect to standard error or standard output to files things get interesting, as you can change where they go. You saw this above when we redirected standard error to `/dev/null`.

Now type these in and try and figure out why they produce different output:

```
command_does_not_exist 2>&1 > outfile
cat outfile
command_does_not_exist > outfile 2>&1
cat outfile
```

Type the above code into the terminal in this lesson.

This is where things get tricky and you need to think carefully!

- **Line 1** will have displayed the error as output and put nothing into the redirected-to `outfile` (**line 2**), while
- **Line 3** did not display the error and captured it in the `outfile` (**line 4**).

Remember that the redirection operator `2>&1` points standard error (file descriptor `2`) at whatever standard output (file descriptor `1`) was pointed to at the time.

If you read the first line carefully, at the point `2>&1` was used, standard output was pointed at the terminal. So standard error is pointed at the terminal from there on.

After that point, standard output is redirected (with the `>` operator) to the file `outfile`.

So at the end of all this:

- The standard error of the output of the command `command_does_not_exist` points at the terminal
- The standard output points at the file `outfile`.

In the second line (`command_does_not_exist > outfile 2>&1`), what is different?

The order of redirections is changed.

Now:

- The standard output of the command `command_does_not_exist` is pointed

- The standard output of the command `command_does_not_exist` is pointed at the file `outfile`

- The redirection operator `2>&1` points file descriptor 2 (standard error) to whatever file descriptor 1 (standard output) is pointed at

So in effect, both standard out and standard error are pointed at the same file (`outfile`).

This pattern of sending all the output to a single file is seen very often, and few understand why it has to be in that order. Once you understand, you will never pause to think about which way round the operators should go again!

## Differences Between Pipes and Redirects #

To recap:

- A pipe passes *standard output* as the *standard input* to another command

- A *redirect* sends a channel of output to a file

A couple of other commonly used operators are worth mentioning here:

```
grep -c file < file1
```

Type the above code into the terminal in this lesson.

The `<` operator redirects standard *input* to the `grep` command from a file. In this case, it is equivalent to the `cat file1 | grep -c file` command you saw earlier.

```
echo line1 > file3
cat file3
echo line2 > file3   # Overwrites file3, replacing its contents with 'line2'
cat file3
echo line3 >> file3  # Appends to file3
cat file3
```

Type the above code into the terminal in this lesson.

- **Lines 1 and 3** above use the `>` operator, while
- **Line 5** uses the `>>` operator.

The `>` operator effectively creates the file anew whether it already exists or not. The `>>` operator, by contrast, *appends* to the end of the file.

As a result, only `line2` and `line3` are added to `file3`.

Pipes and Redirects Quiz

1 Choose the true statements

COMPLETED 0%

1 of 2

## What You Learned

- What *file redirection* is
- What *pipes* do
- The differences between *standard output* and *standard error*
- How to *redirect standard output* to the same location as *standard error* (and vice versa)
- How to *redirect standard output or standard error* (or both) to a file

## What Next? #

You’ve completed the ‘Core Bash’ section! Next you’ll move onto the ‘Scripting Bash’ section, where you’ll learn about the tools needed to write significant and useful bash scripts.

## Exercises #

1) Try a few different commands and work out what output goes to standard output and what output goes to standard error. Try triggering errors by misusing programs.

2) Write commands to redirect standard output to file descriptor `3`.

3) Research what the `|&` pipe operator does. If you can’t find out, try the command `cat file2 |& grep -c file` and see what changes compared to when you ran it above.