Piping and Redirection!

Keeping the data flowing

Introduction

Learn how easy it is to use piping and redirection to create powerful workflows that will automate your work, saving you time and effort.

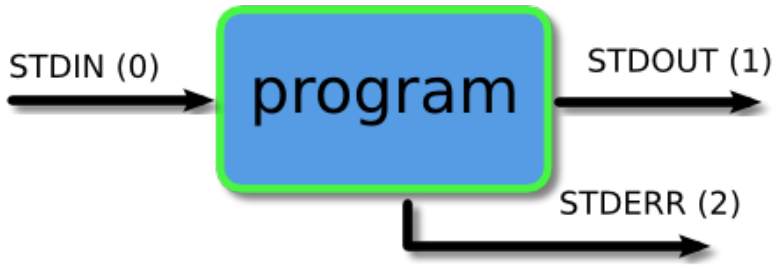
In the previous two sections we looked at a collection of filters that would manipulate data for us. In this section we will see how we may join them together to do more powerful data manipulation.

There is a bit of reading involved in this section. Even though the mechanisms and their use are quite simple, it is important to understand various characteristics about their behaviour if you wish to use them effectively.

So what are they?

Every program we run on the command line automatically has three data streams connected to it.

- STDIN (0) - Standard input (data fed into the program)
- STDOUT (1) - Standard output (data printed by the program, defaults to the terminal)
- STDERR (2) - Standard error (for error messages, also defaults to the terminal)



Piping and redirection is the means by which we may connect these streams between programs and files to direct data in interesting and useful ways.

We'll demonstrate piping and redirection below with several examples but these mechanisms will work with every program on the command line, not just the ones we have used in the examples.

Redirecting to a File

Normally, we will get our output on the screen, which is convenient most of the time, but sometimes we may wish to save it into a file to keep as a record, feed into another system, or send to someone else. The greater than operator (>) indicates to the command line that we wish the programs output (or whatever it sends to STDOUT) to be saved in a file instead of printed to the screen. Let's see an example.

Terminal

```
1.  user@bash: ls
2.  barry.txt bob example.png firstfile foo1 video.mpeg
3.  user@bash: ls > myoutput
4.  user@bash: ls
5.  barry.txt bob example.png firstfile foo1 myoutput video.mpeg
6.  user@bash: cat myoutput
7.  barry.txt
8.  bob
9.  example.png
10. firstfile
11. foo1
12. myoutput
13. video.mpeg
14. user@bash:
```

Let's break it down:

- **Line 1** Let's start off by seeing what's in our current directory.
- **Line 3** Now we'll run the same command but this time we use the > to tell the terminal to save the output into the file myoutput. You'll notice that we don't need to create the file before saving to it. The terminal will create it automatically if it does not exist.
- **Line 4** As you can see, our new file has been created.
- **Line 6** Let's have a look at what was saved in there.

Some Observations

You'll notice that in the above example, the output saved in the file was one file per line instead of all across one line when printed to the screen. The reason for this is that the screen is a known width and the program can format its output to suit that. When we are redirecting, it may be to a file, or it could be somewhere else, so the safest option is to format it as one entry per line. This also allows us to easier manipulate that data later on as we'll see further down the page.

Tip

When piping and redirecting, the actual data will always be the same, but the formatting of that data may be slightly different to what is normally printed to the screen. Keep this in mind.

You'll also notice that the file we created to save the data into is also in our listing. The way the mechanism works, the file is created first (if it does not exist already) and then the program is run and output saved into the file.

Saving to an Existing File

If we redirect to a file which does not exist, it will be created automatically for us. If we save into a file which already exists, however, then it's contents will be cleared, then the new output saved to it.

Terminal

```
user@bash: cat myoutput
barry.txt
bob
example.png
firstfile
foo1
myoutput
video.mpeg
user@bash: wc -l barry.txt > myoutput
user@bash: cat myoutput
7 barry.txt
user@bash:
```

We can instead get the new data to be appended to the file by using the double greater than operator (>>).

Terminal

```
user@bash: cat myoutput
7 barry.txt
user@bash: ls >> myoutput
user@bash: cat myoutput
7 barry.txt
barry.txt
bob
example.png
firstfile
foo1
myoutput
video.mpeg
user@bash:
```

Redirecting from a File

If we use the less than operator (<) then we can send data the other way. We will read data from the file and feed it into the program via it's STDIN stream.

Terminal

```
user@bash: wc -l myoutput
8 myoutput
user@bash: wc -l < myoutput
```

```
4. 8
5. user@bash:
```

A lot of programs (as we've seen in previous sections) allow us to supply a file as a command line argument and it will read and process the contents of that file. Given this, you may be asking why we would need to use this operator. The above example illustrates a subtle but useful difference. You'll notice that when we ran wc supplying the file to process as a command line argument, the output from the program included the name of the file that was processed. When we ran it redirecting the contents of the file into wc the file name was not printed. This is because whenever we use redirection or piping, the data is sent anonymously. So in the above example, wc recieved some content to process, but it has no knowledge of where it came from so it may not print this information. As a result, this mechanism is often used in order to get ancillary data (which may not be required) to not be printed.

We may easily combine the two forms of redirection we have seen so far into a single command as seen in the example below.

Terminal

```
1. user@bash: wc -l < barry.txt > myoutput
2. user@bash: cat myoutput
3. 7
4. user@bash:
```

Redirecting STDERR

Now let's look at the third stream which is Standard Error or STDERR. The three streams actually have numbers associated with them (in brackets in the list at the top of the page). STDERR is stream number 2 and we may use these numbers to identify the streams. If we place a number before the > operator then it will redirect that stream (if we don't use a number, like we have been doing so far, then it defaults to stream 1).

Terminal

```
1. user@bash: ls -l video.mpg blah.foo
2. ls: cannot access blah.foo: No such file or directory
3. -rwxr--r-- 1 ryan users 6 May 16 09:14 video.mpg
4. user@bash: ls -l video.mpg blah.foo 2> errors.txt
5. -rwxr--r-- 1 ryan users 6 May 16 09:14 video.mpg
6. user@bash: cat errors.txt
7. ls: cannot access blah.foo: No such file or directory
8. user@bash:
```

Maybe we wish to save both normal output and error messages into a single file. This can be done by redirecting the STDERR stream to the STDOUT stream and redirecting STDOUT to a file. We redirect to a file first then redirect the error stream. We identify the redirection to a stream by placing an & in front of the stream number (otherwise it would redirect to a file called 1).

Terminal

```
1. user@bash: ls -l video.mpg blah.foo > myoutput 2>&1
2. user@bash: cat myoutput
```

```
ls: cannot access blah.foo: No such file or directory
-rwxr--r-- 1 ryan users 6 May 16 09:14 video.mpg
user@bash:
```

Piping

So far we've dealt with sending data to and from files. Now we'll take a look at a mechanism for sending data from one program to another. It's called piping and the operator we use is (|) (found above the backslash (\ ) key on most keyboards). What this operator does is feed the output from the program on the left as input to the program on the right. In the example below we will list only the first 3 files in the directory.

Terminal

```
user@bash: ls
barry.txt bob example.png firstfile foo1 myoutput video.mpeg
user@bash: ls | head -3
barry.txt
bob
example.png
user@bash:
```

We may pipe as many programs together as we like. In the below example we have then piped the output to tail so as to get only the third file.

Terminal

```
user@bash: ls | head -3 | tail -1
example.png
user@bash:
```

Tip

Any command line arguments we supply for a program must be next to that program.

Tip

I often find people try and write their pipes all out in one go and make a mistake somewhere along the line. They then think it is in one point but in fact it is another point. They waste a lot of time trying to fix a problem that is not there while not seeing the problem that is there. If you build your pipes up incrementally then you won't fall into this trap. Run the first program and make sure it provides the output you were expecting. Then add the second program and check again before adding the third and so on. This will save you a lot of frustration.

You may combine pipes and redirection too.

Terminal

```
user@bash: ls | head -3 | tail -1 > myoutput
user@bash: cat myoutput
example.png
user@bash:
```

More Examples

Below are some more examples to give an idea of the sorts of things you can do with piping. There are many things you can achieve with piping and these are just a few of them. With experience and a little creative thinking I'm sure you'll find many more ways to use piping to make your life easier.

All the programs used in the examples are programs we have seen before. I have used some command line arguments that we haven't covered yet however. Look up the relevant man pages to find out what they do. Also you can try the commands yourself, building up incrementally to see exactly what each step is doing.

In this example we are sorting the listing of a directory so that all the directories are listed first.

Terminal

```
user@bash: ls -l /etc | tail -n +2 | sort
drwxrwxr-x 3 nagios nagcmd 4096 Mar 29 08:52 nagios
drwxr-x--- 2 news news 4096 Jan 27 02:22 news
drwxr-x--- 2 root mysql 4096 Mar 6 22:39 mysql
...
user@bash:
```

In this example we will feed the output of a program into the program less so that we can view it easier.

Terminal

```
user@bash: ls -l /etc | less
(Full screen of output you may scroll. Try it yourself to see.)
```

Identify all files in your home directory which the group has write permission for.

Terminal

```
user@bash: ls -l ~ | grep '^.....w'
drwxrwxr-x 3 ryan users 4096 Jan 21 04:12 dropbox
user@bash:
```

Create a listing of every user which owns a file in a given directory as well as how many files and directories they own.

Terminal

```
user@bash: ls -l /projects/ghosttrail | tail -n +2 | sed 's/\s\s*/ /g' | cut -d ' ' -f
3 | sort | uniq -c
8 anne
34 harry
```

```
37 tina
18 ryan
user@bash:
```

Summary

Stuff We Learnt

>
Save output to a file.

>>
Append output to a file.

<
Read input from a file.

2>
Redirect error messages.

|
Send the output from one program as input to another program.

Important Concepts

Streams
Every program you may run on the command line has 3 streams, STDIN, STDOUT and STDERR.

Activities

Let's mangle some data:

- First off, experiment with saving output from various commands to a file. Overwrite the file and append to it as well. Make sure you are using a both absolute and relative paths as you go.
- Now see if you can list only the 20th last file in the directory /etc.
- Finally, see if you can get a count of how many files and directories you have the execute permission for in your home directory.