2.7.5 Redirection and Piping Facts

Redirection and piping are two Linux shell features that allow the input and output of a command to be a file or another command.

This lesson covers the following topics:

- Standard streams and bash shell file descriptors
- Redirection
- Piping
- The tee command
- Here documents
- Device files often used with redirection and piping

Standard Streams and Bash Shell File Descriptors

The bash shell maintains three standard data streams, which are preconnected input and output communication channels. Unless configured otherwise, bash commands use the following standard streams:

- Standard input (stdin) is data that is typically streamed from the keyboard. If a bash command accepts input, by default, it is gathered from stdin.
- Standard output (stdout) is data that is typically streamed to the console screen. By default, bash commands send their output to stdout.
- Standard error (stderr) is data that is also typically streamed to the console screen. If a command needs to output an error messages or give diagnostics, by default, it sends this output to stderr.

The bash shell assigns a file descriptor to each of the standard streams. A file descriptor is a handle or number that identifies an open file or other data source and how that resource can be accessed. The following table summarizes these ideas.

Standard Stream	File Descriptor	Associated Device
stdin	0	Keyboard
stdout	1	Console screen or graphical shell window
stderr	2	Console screen or graphical shell window

Redirection

Linux commands can be modified to divert the standard input, output, and error streams to locations other than the default. This process is call redirection and is implemented using the following command operators.

Command Operator	Description
> and **1>**	Redirects command output that is normally sent to stdout to the file name that follows the operator. The 1 is implied so that **>** and **1>** are functionally identical. • If the file that follows the operator exists, it is overwritten. • If the file doesn't exist, it is created. • If there is no output generated by the command, the file will be empty.
2>	Redirects command output that is normally sent to stderr to the file name that follows the operator. The '2' is mandatory. • If the file that follows the operator exists, it is overwritten. • If the file doesn't exist, it is created. • If there is no error generated by the command, the file will be empty.
>>, **1>>** and **2>>**	The **>>** operator functions in the same way as the **>** operator except that any output/errors are appended to the file that follows the operator. • The **>>** and **1>>** operators appends the output sent to stdout. • The **2>>** operator appends the output sent to stderr. • If the file that follows the operator exists, it is appended with the output/error. • If the file doesn't exist, it is created. • The file will be not be appended if there is no output/errors generated by the command.
&>	Redirects both command output that is normally sent to stdout and command errors that are normally sent to stderr to the file name that follows the operator. • As part of a command, **&> myfile.txt** is equivalent to **> myfile.txt 2> &1** or **> myfile.txt 2> myfile.txt**. • File creation follows the rules for the **>** operator.
&>>	The **&>>** operator functions in the same way as the **&>** operator except that both output and errors are appended to the file that follows the operator. • As part of a command, **&>> myfile.txt** is equivalent to **>> myfile.txt 2>> &1** (where **&** indicates that what follows is a file descriptor and not a filename) or **>> myfile.txt 2>> myfile.txt**.

	• File creation follows the rules for the **>** operator.
< and **0<**	Redirects command input that is normally read from stdin so that it is read from the file name that follows the operator. The 0 is implied so that **>** and **0>** are functionally identical. • If the file that follows the operator exists, it is used as input. • If the file doesn't exist, an error is shown. • If there is no input needed by the command, the file is ignored.

The following examples demonstrate redirection concepts.

Example	Result
ls /usr > /tmp/deleteme	Writes the list of files in the /usr directory to a file named /tmp/deleteme.
ls /nonesuch > /tmp/deleteme	Creates an empty /tmp/deleteme file (or overwrites it as an empty file if it exists) and displays the error message '/nonesuch not found' (because the /nonesuch directory does not exist).
ls /nonesuch 2> /tmp/deleteme	Writes the error message '/nonesuch not found' (because the /nonesuch directory does not exist) a file named /tmp/deleteme. The /tmp/deleteme file will be overwritten if it already exists.
ls /bin /nonesuch > /tmp/deleteme	Writes a listing of the files and directories within the /bin directory to the /tmp/deleteme file and displays the error message */nonesuch not found* to the console screen or shell window. The /tmp/deleteme file will be overwritten if it already exists.
ls /bin /nonesuch > /tmp/deleteme 2>&1	Writes a listing of the files and directories within the /bin directory to the /tmp/deleteme file and also writes the error message */nonesuch not found* to the /tmp/deleteme file. Both the list of files and directories within the /bin directory and the error message are written to the same file.
ls /bin >> /tmp/deleteme	Appends the list of files from the /bin directory to the /tmp/deleteme file. The /tmp/deleteme file will be created if it does not exist.
sort < unordered.txt > ordered.txt	Uses the contents of the unordered.txt as input to the **sort** command, and then writes the sorted contents to the file named ordered.txt. If the ordered.txt file already exists, it will be overwritten.

Piping

Piping redirects the output from one command to be the input of another command.

- A Linux pipe is represented by a vertical bar (**|**).

- The pipe functionality is similar to that of using stdout redirection to write the output of one command to an intermediate file that is then used as input to a second command using stdin redirection.
- A plumbing pipe, where water enters from one end and exits the other, can be used as a mnemonic to help explain how the pipe (|) operator works.

The following examples demonstrate piping concepts.

Example	Result
ls /bin | sort | mail jdoe	Takes the list of the contents of **/bin** directory, sorts it, and then mails the sorted contents to jdoe.
cat /usr/wordlist1 /usr/wordlist2 | sort > sortedwordlist	Pipes the combined contents of the **wordlist1** and **wordlist2** files to sort and then redirect the sorted output to the **sortedwordlist** file.

The tee Command

There may be times when you want to view the output of a command as it is normally sent to the console screen (stdout), but you also want the same output to be saved in a file. This can be done using the **tee** command.

- The output from a command is piped to the **tee** command.
- The file used to store the output is added as a **tee** command argument.
- A plumbing tee, where water flow is divided from one pipe to two separate pipes, can help you understand how the **tee** command works.

The following examples demonstrate **tee** command concepts.

Example	Result
ls /bin | tee binfiles.txt	Displays the files and directories contained in the **/bin** directory on the console screen (or shell window) and writes the same information to the **binfiles.txt** file.
ls -1 *.txt | wc -l | tee count.txt	Pipes a one-column list of files that end with *.txt* in the current directory to the **wc** (word count) command and then take that output, which gives the number of files in the list, and pipes that to the tee command that displays this number on the console screen (or shell window) and writes the same number to the **count.txt** file.

Here Documents

A here document is a block of text that is redirected as input to a command. Here documents are often used in shell scripts.

- A command is followed by the **<<** operator, which is then followed by a marker, which is traditionally an uppercase word.

  > The term *here document* may have origins in the practice of using the word *HERE* as the marker.

- Lines of text are included in the block.
- The end of the block of text is indicated by the same marker that follows the **<<** operator.
- The shell passes the block of text to the command as input.

The following examples demonstrate here document concepts.

Example	Explanation
cat << HERE > Today, we hope you are > learning a great deal > about Linux redirection > and piping from TestOut. >HERE	The **cat <<** syntax, when executed in a shell by a user, is interactive. Each line is a separate entry with an enter command to go to the next line. The full text is taken together after the designated string, in this case **HERE**. The following lines are displayed on the console screen or shell window: Today, we hope you are learning a great deal about Linux redirection and piping from TestOut. This may be useful when creating shell scripts that present explanations and documentation to users.
lftp machine -uUser,Passwd <<END cd your_dir get your_file bye END	Multiple commands are entered during an ftp session. These commands are managed using the lftp program. This may be helpful when automating a long list of ftp commands.

Device Files Often Used with Redirection and Piping

Device files are file-like access point to hardware devices. There are two device files that are often used with redirection and piping. /dev/tty and /dev/null.

Device	Description

File	
/dev/tty	The first terminals were Teletype (abbreviated as *tty*), which can be compared to a remote controlled typewriter. The /dev/tty device file is associated with the computer's controlling terminal or the shell's window. • Data can be both written to and read from this file. • Text written to this file is displayed on the console monitor's screen or shell window. • Text read from this file originates from the console's keyboard. • The **/dev/tty** device file is similar to a combination of stdin and stdout. Both stdin and stdout are accessed as data streams, whereas **/dev/tty** is accessed like a file.
/dev/null	The /dev/null device file is associate with a null device. A null device is commonly used for disposing unwanted output streams. • While a command can read from **/dev/null**, commands typically write unwanted output or unwanted error messages to **/dev/null**. • A slang word for the **/dev/null** device file is bit bucket.

The following demonstrates the use of the **/dev/tty** and **/dev/null** device files.

Example	Explanation
echo "test" > /dev/tty	Writes the word *test* to the console or to a shell windows. This is redundant, since the **echo** command by itself performs the same action.
sort < /dev/tty > sortkeyboard.txt	The text entered using the keyboard is sorted and written to the sortkeyboard.txt file. The **< /dev/tty** operation will continue to accept keyboard input until the user enters an end-of-file (EOF) sequence using the Ctrl+D key combination.
rm deleteme.txt 2> /dev/null	Deletes the **deleteme.txt** file if it exists. If it doesn't exist, don't display an error message. This logic is often used in shell scripts to suppress error messages that are not important to the script's overall purpose.