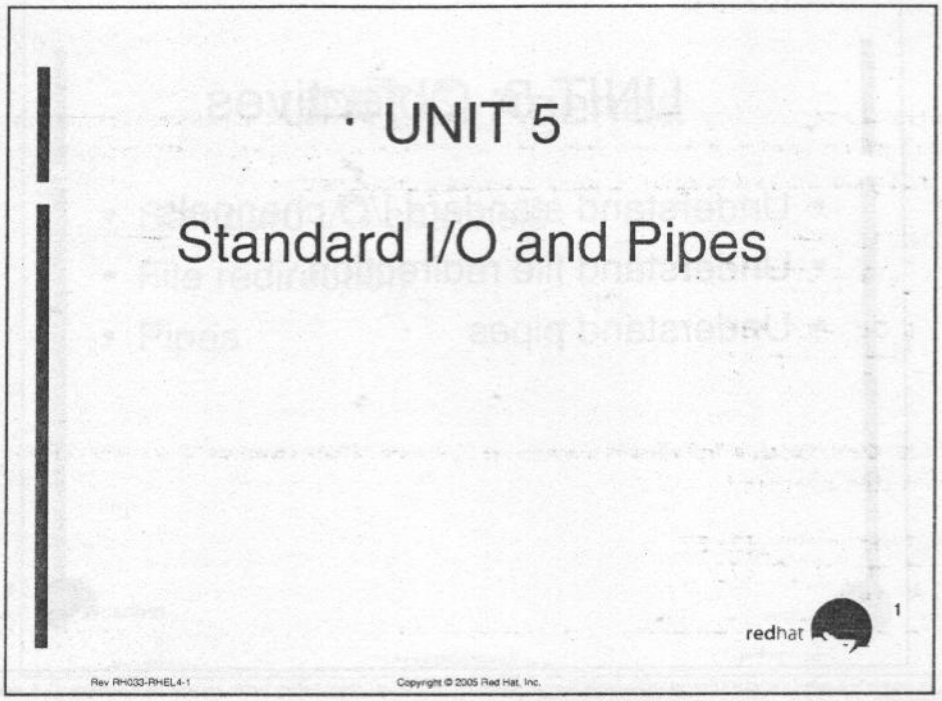
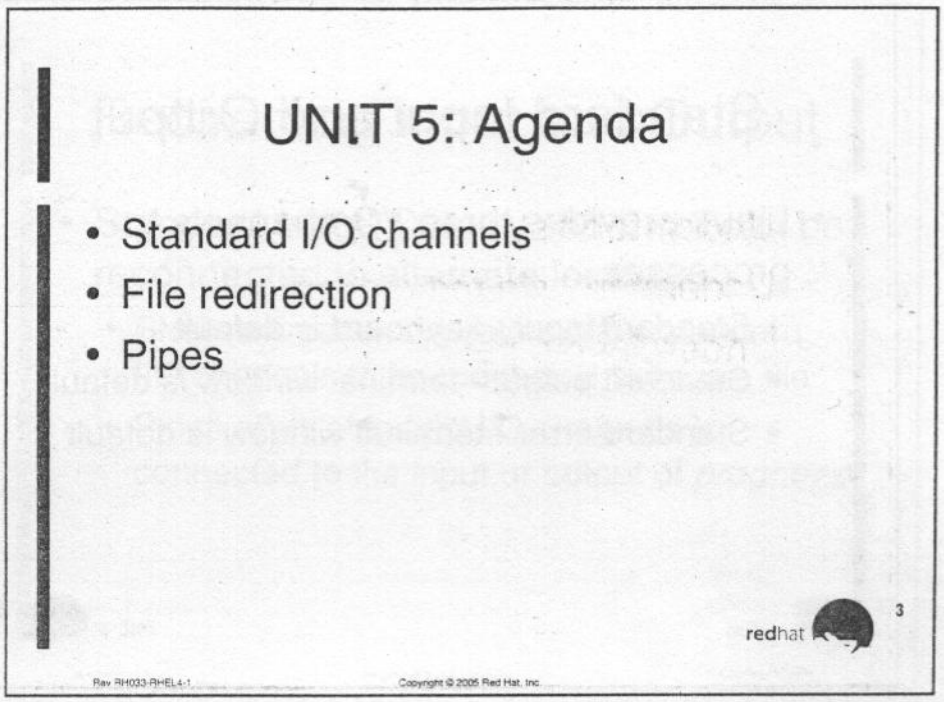
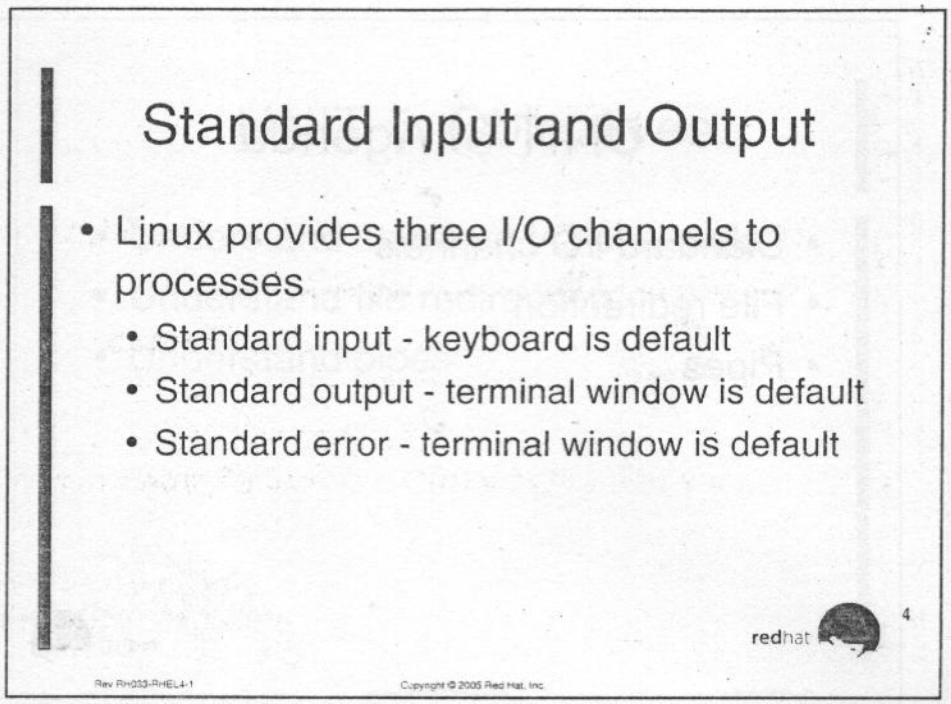


UNIT 5: Objectives

- Understand standard I/O channels
- Understand file redirection
- Understand pipes





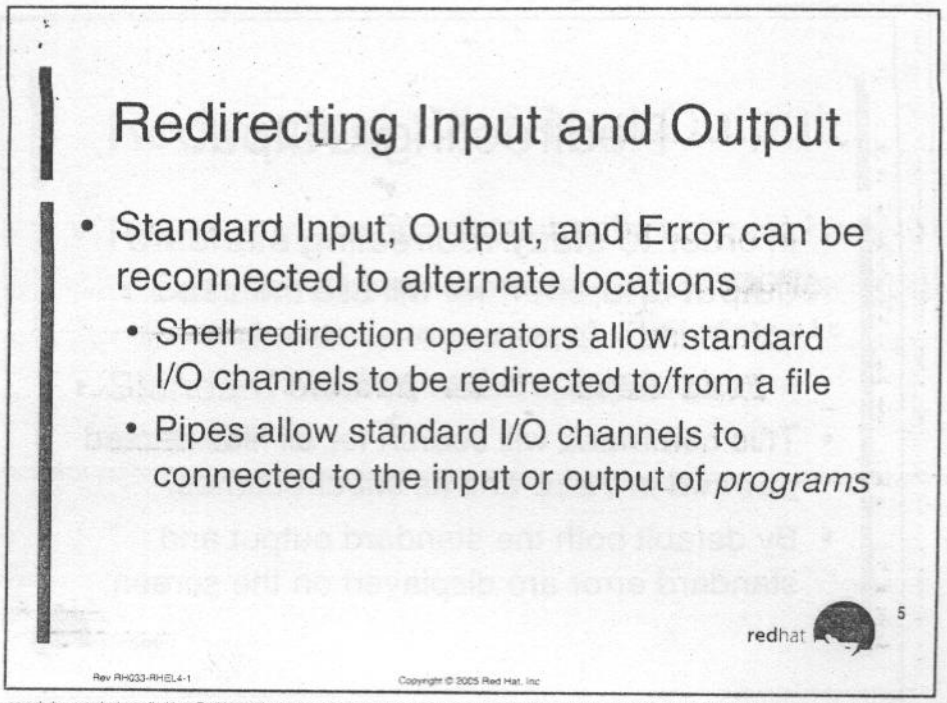
One of the most important features of Linux (and UNIX) is the streaming nature of data known as standard input, standard output, and standard error . In general, this allows the input from a program to come from any source, and the output to go to any source. In addition, the output from one command can be fed directly to the input of another command.

Standard output, by default, is the screen or terminal window. Most commands send their output to standard output without explicitly being told to do so. When standard output is sent to a destination other than the screen, such as a file, one is *redirecting standard output*.

Standard input defaults to the keyboard. Most commands use files as their source of input, but will accept standard input from other sources, via redirection.

A third data stream is standard error. This is a secondary output stream that carries warnings, usage messages, error messages and other "out-of-band" information, in an output stream distinct from standard output. This error stream is normally sent to the screen, but may be redirected elsewhere.

These streams are abbreviated as stdin (called file descriptor number 0), stdout (file descriptor number 1), and stderr (file descriptor number 2). The fact that there are two output channels allows separation of error messages from normal output. For example, error messages could be saved in a file with the normal output going to the monitor.



The standard output of commands, which ordinarily displays on the terminal, can be redirected into a file or piped into another command.

Standard error, which also ordinarily displays on the terminal, can be redirected into a file. Although it is also possible to pipe standard error into a file using some fairly complex syntax, this is generally not done.

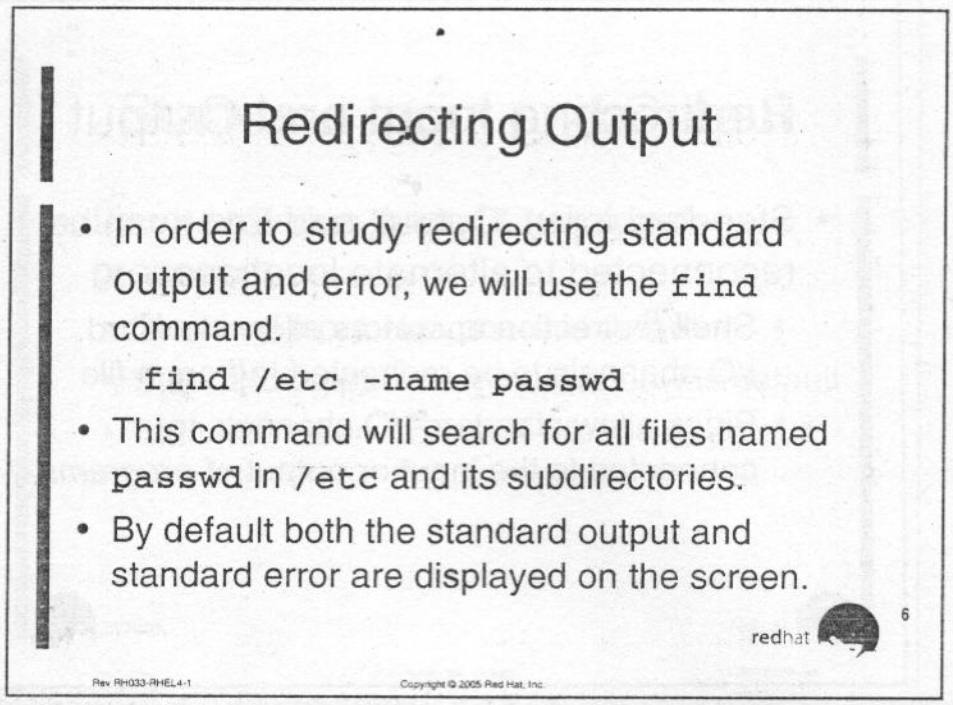
Standard input, ordinarily coming from the keyboard, can be redirected from a file. More commonly, the standard output of one command can be piped into the standard input of another command.

Common Redirection Operators

`>`	`command > file`	Directs standard output of *`command`* to *`file`*
`>>`	`command >> file`	Appends standard output of *`command`* to *`file`*
`<`	`command < file`	*`command`* receives its input from *`file`*
`2>`	`command 2> file`	Error messages from *`command`* are directed to *`file`*
`2>>`	`command 2>> file`	Errors messages from *`command`* are appended to *`file`*

Piping

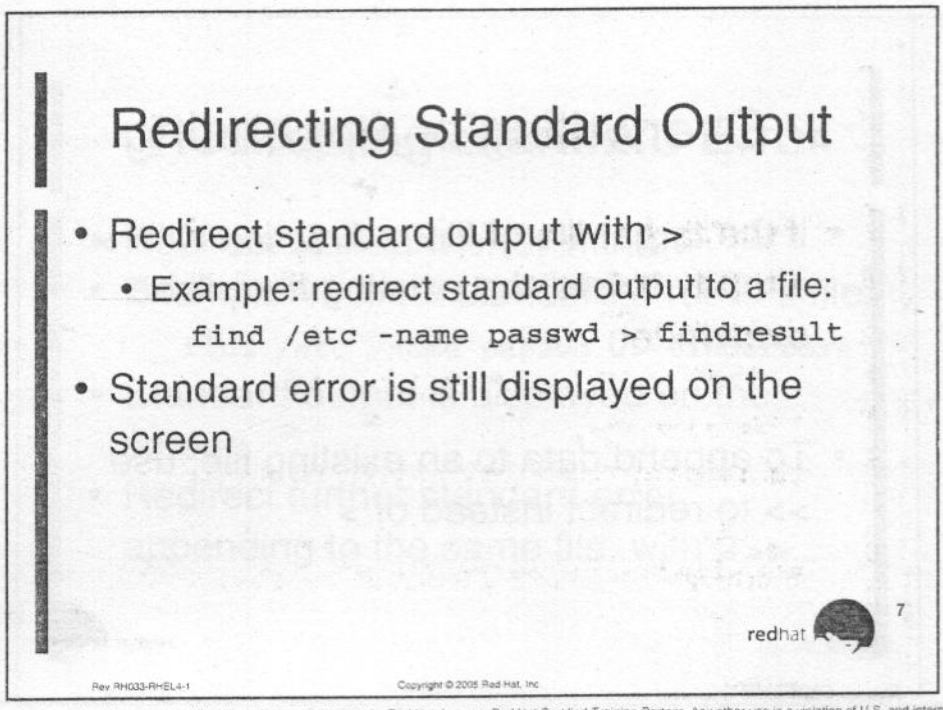
`|`	`command1 | command2`	Pipes the standard output of command1 into the standard input of command2



Example output

```
[bob@station2 tmp]$ find /etc -name passwd
/etc/passwd
find: /etc/default: Permission denied
/etc/pam.d/passwd
```

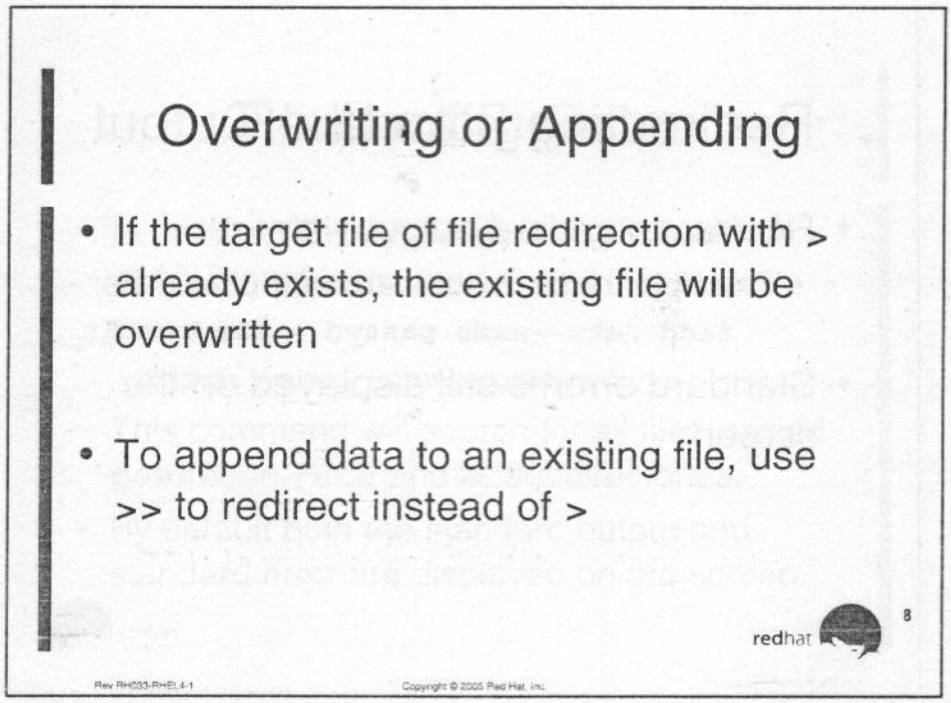
The specific error messages you receive may vary, depending on the particular packages you have installed.



Example output with stdout redirected to a file and stderr displayed to the terminal:

```
[bob@station2 tmp]$ find /etc -name passwd > findresult
find: /etc/default: Permission denied

[bob@station2 tmp]$ cat findresult
/etc/passwd
/etc/pam.d/passwd
```



Redirect output and create/overwrite the `output` file

```
[bob@station2 tmp]$ find /etc -name passwd > output
find: /etc/default: Permission denied
```

Redirect output and append to the `output` file

```
[bob@station2 tmp]$ find /etc -name passwd >> output
find: /etc/default: Permission denied
find: /etc/cups/certs: Permission denied
```

View the `output` file

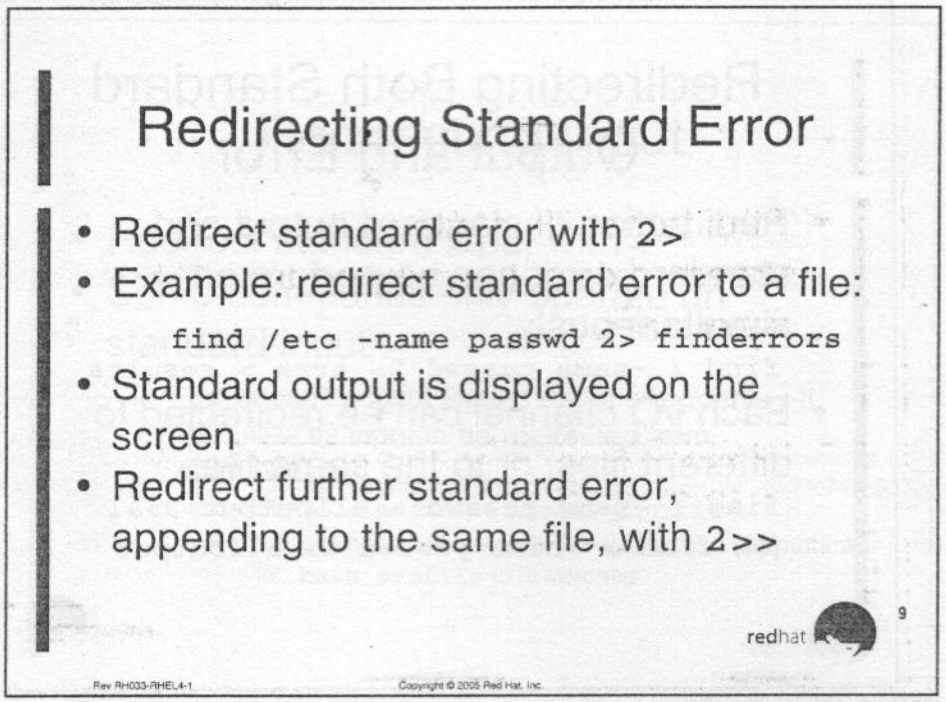
```
[bob@station2 tmp]$ cat output
/etc/passwd
/etc/pam.d/passwd
/etc/passwd
/etc/pam.d/passwd
```

Redirect output and overwrite the `output` file

```
[bob@station2 tmp]$ find /etc -name passwd > output
find: /etc/default: Permission denied
```

View the `output` file

```
[bob@station2 tmp]$ cat output
/etc/passwd
/etc/pam.d/passwd
```



Redirect stderr to the `finderrors` file

```
[bob@station2 tmp]$ find /etc -name passwd 2> finderrors
/etc/passwd
/etc/pam.d/passwd
```

View the `finderrors` file

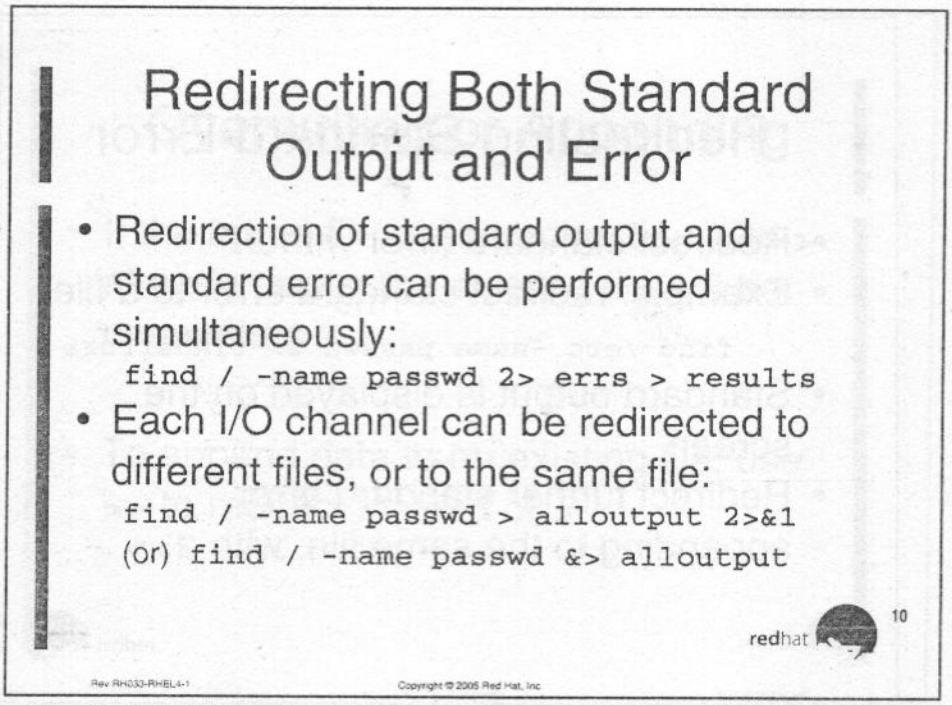
```
[bob@station2 tmp]$ cat finderrors
find: /etc/default: Permission denied
find: /etc/cups/certs: Permission denied
```

Redirect some more stderr to the `finderrors` file

```
[bob@station2 tmp]$ find /tmp -name passwd 2>> finderrors
```

View the `finderrors` file

```
[bob@station2 tmp]$ cat finderrors
find: /etc/default: Permission denied
find: /tmp/orbit-root: Permission denied
```



Redirect stderr to the `errs` file and redirect stdout to the `results` file

```
[bob@station2 tmp]$ find /etc -name passwd 2> errs > results
```

View the `errs` file

```
[bob@station2 tmp]$ cat errs
find: /etc/default: Permission denied
```

View the `results` file

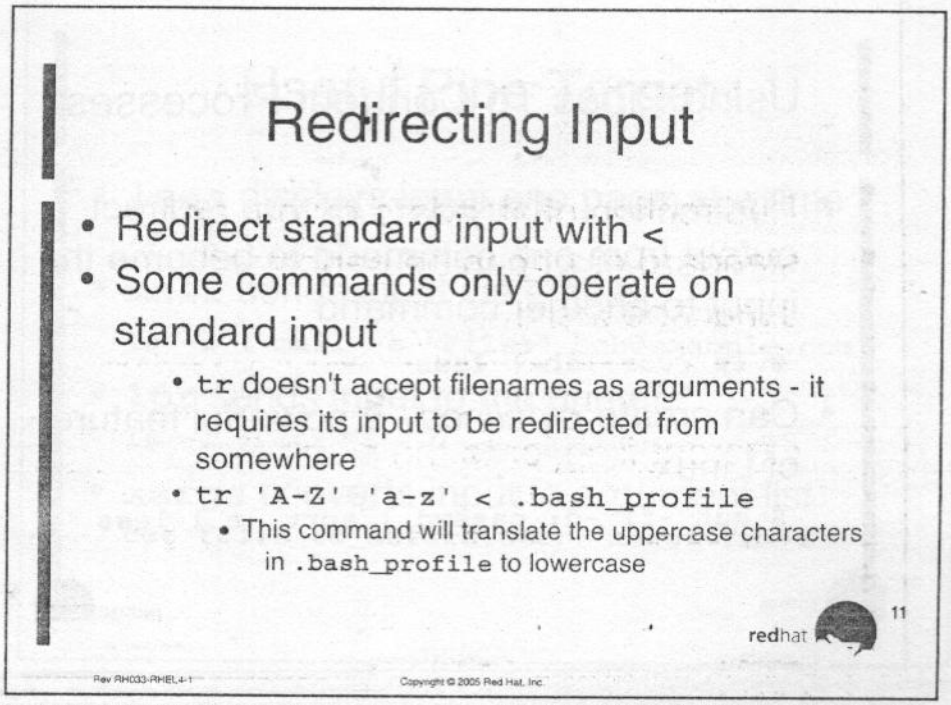
```
[bob@station2 tmp]$ cat results
/etc/passwd
/etc/pam.d/passwd
```

Redirect both stderr and stdout to the `alloutput` file

```
[bob@station2 tmp]$ find /etc -name passwd > alloutput 2>&1
```

View the `alloutput` file

```
[bob@station2 tmp]$ cat alloutput
/etc/passwd
find: /etc/default: Permission denied
/etc/pam.d/passwd
```



Many Linux commands, like `cat`, will take their input from a file if one is given as an argument. Commands such as these usually accept either a filename or standard input. The following two commands produce the same output:

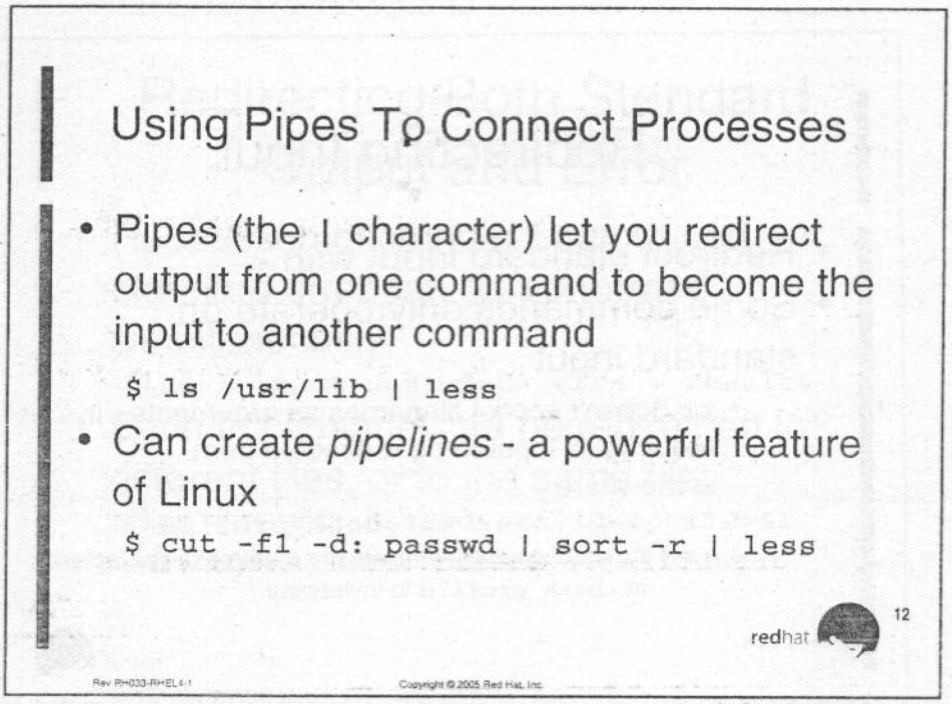
```
$ cat filename.txt
$ cat < filename.txt
```

A few commands, like the character translator `tr`, do not accept filenames as arguments and must have their input redirected.

Example output

```
[bob@station2 tmp]$ cat textfile.txt
TEXT TYPED IN ALL CAPITALS IS OFTEN DIFFICULT TO READ.
TRANSLATING SUCH TEXT INTO LOWER CASE CAN IMPROVE
ITS LEGIBILITY.

[bob@station2 tmp]$ tr 'A-Z' 'a-z' < textfile.txt
text typed in all capitals is often difficult to read.
translating such text into lower case can improve
its legibility.
```



Two of the basic tenets of UNIX philosophy are: make small programs that do one thing well, and expect the output of every program to become the input to another, as yet unknown, program. The use of pipes let you leverage the effects of these two design principles to create extremely powerful chains of commands to achieve your desired result. Most command-line programs that operate on files can also accept input on standard in, and output to standard out. Some commands like `tr` are designed to only operate within a pipe (can not operate on files directly).

Any command that writes to standard output can be used on the left-hand side of a pipe.

Any command that reads from standard input can be used on the right-hand side of a pipe.

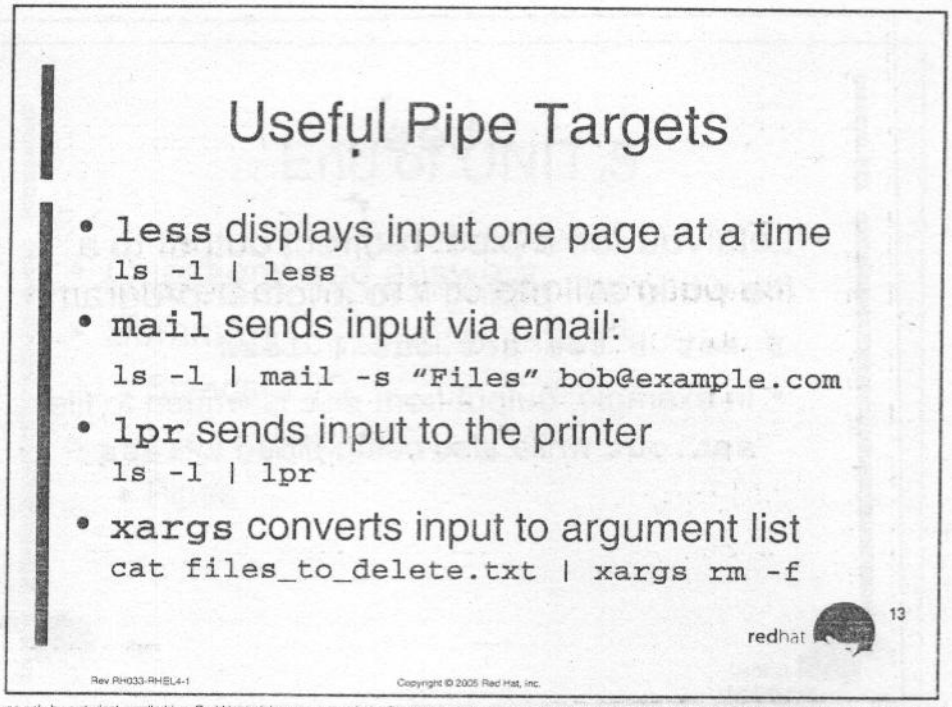
Multiple commands can be chained together with pipes.

Example output

```
[bob@station2 man]$ pwd
/usr/share/man

[bob@station2 man]$ ls -C | tr 'a-z' 'A-Z'
MAN1   MAN3   MAN5   MAN7   MAN9   PT_BR    WHATIS
MAN2   MAN4   MAN6   MAN8   MANN   TMAC.H
```

In the example above, lower case letters of the alphabet from the file listing are converted to upper case letters.



Earlier we demonstrated using the `less` command to display a text file one screenfull at a time. But what if the output of a command takes up more than a screen? You can also use `less` in these situations to buffer and view the command's output by piping the command into it.

The traditional mail command is rather old fashioned for a mailer. But, it is extremely useful for mailing the standard output of a command. The -s option allows you to specify a subject for the email.

You can also pipe output to the lpr command. This will send the output to your system's default printer. If you want to use a non-default printer you can specify one with the `-P` option:

```
ls -l | lpr -P some_printer
```

All of the above examples demonstrate sending the output of one command to the standard input of another command. But what if you wanted to convert the output of one command into part of the command line for another? For example, in the example shown in this slide we have a file called `files_to_delete.txt` that contains a list of filenames. Suppose it contained the following files:

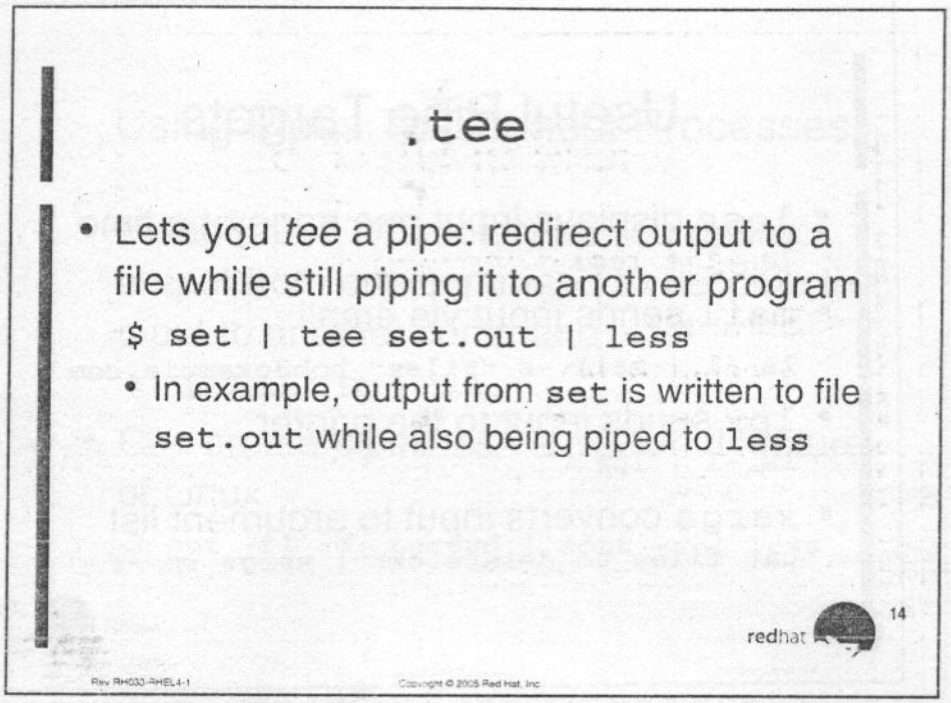
/home/student/letter.txt

/home/student/images/pic.jpg

/tmp/foo

The command shown in the slide would combine the lines of this file into a space-delimited list of arguments to the `rm -f` command. In other words, the command would be equivalent to running:

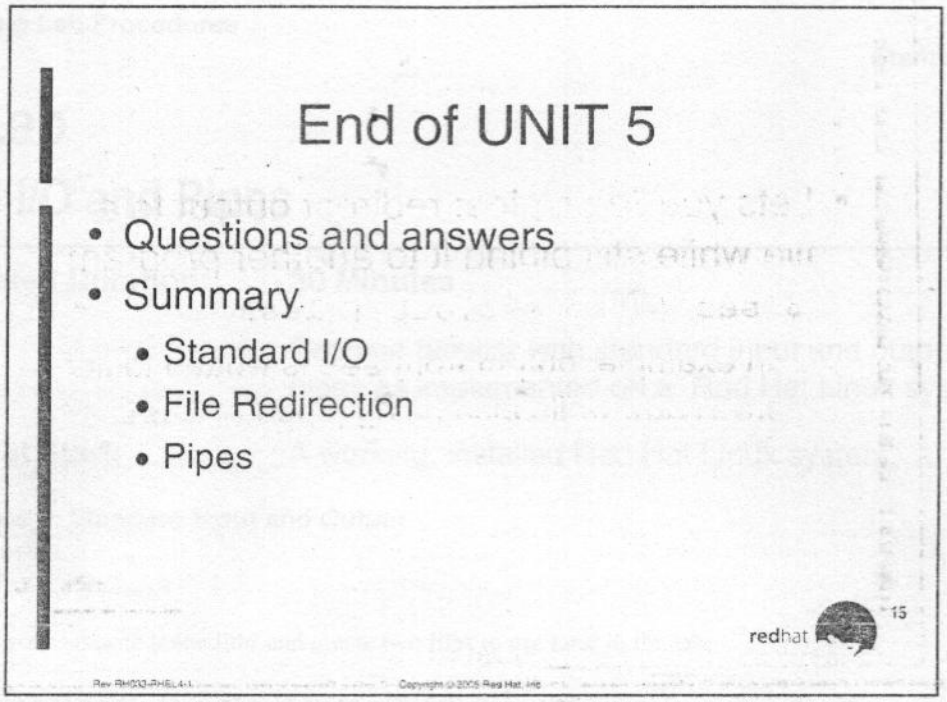
```
rm -f /home/student/letter.txt /home/student/images/pic.jpg /tmp/foo
```



Uses of `tee`

`tee` is useful to save the output at various stages from a long sequence of pipes. For example:

```
$ command | tee stage1.out | sort | tee stage2.out | uniq -c | \
  tee stage3.out | sort -r | tee stage4.out | less
```



Important files covered in this Unit:

Important commands covered in this Unit:

`tee`