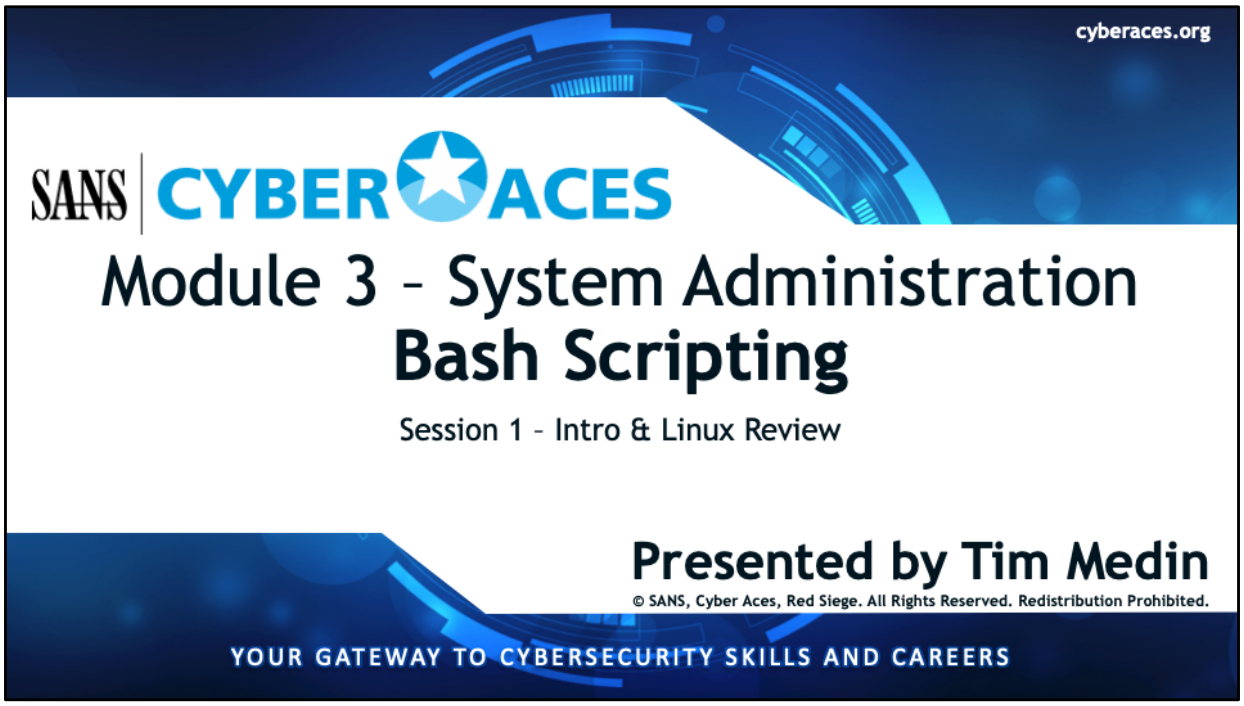
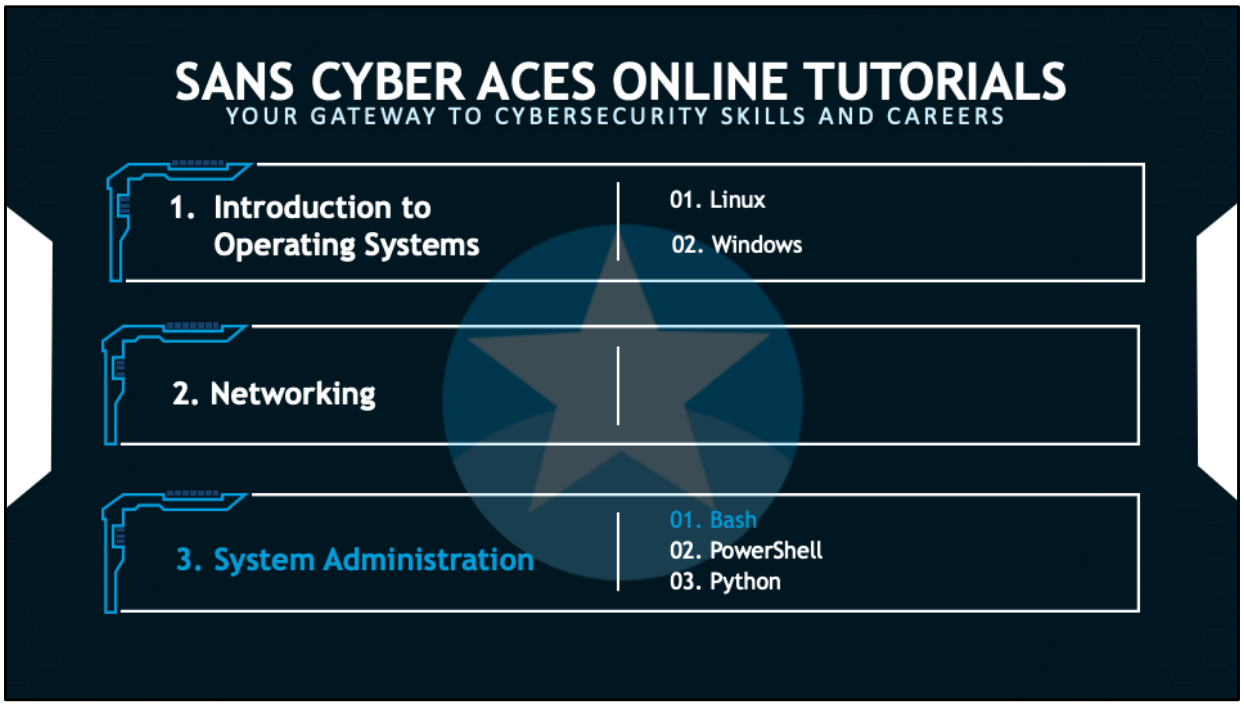


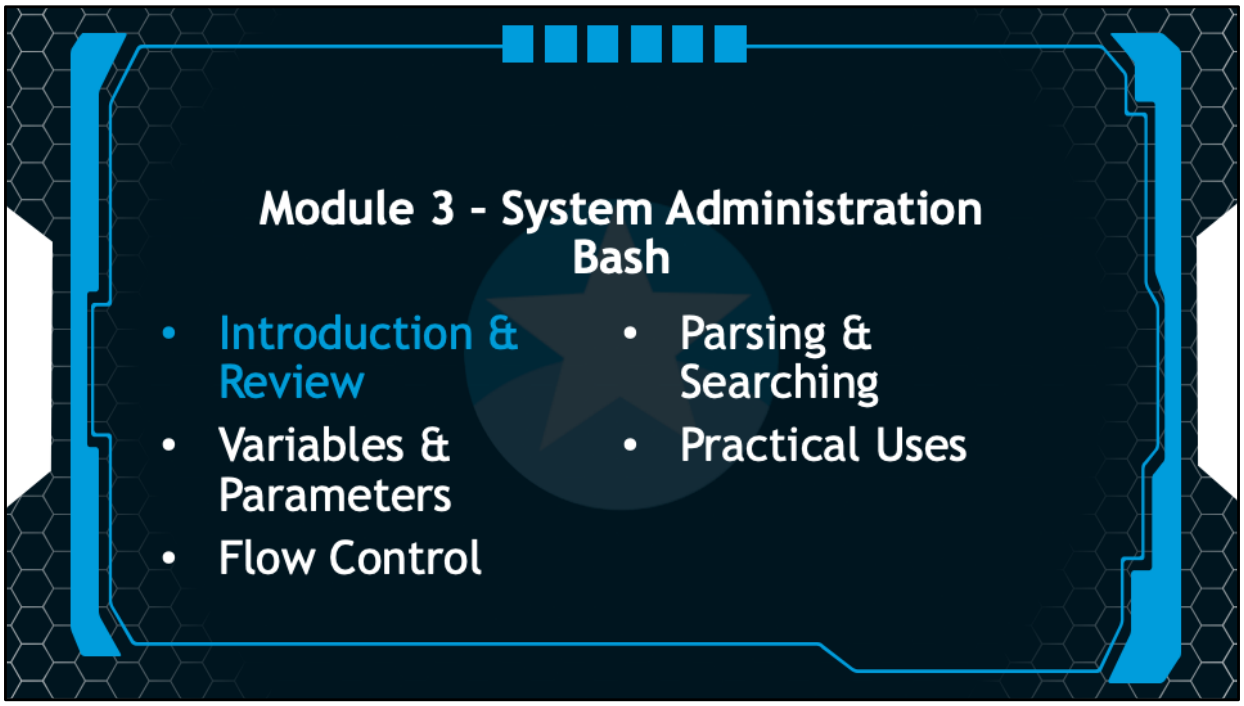
Welcome to Cyber Aces, Module 3! This module provides an introduction to the Bash Scripting.



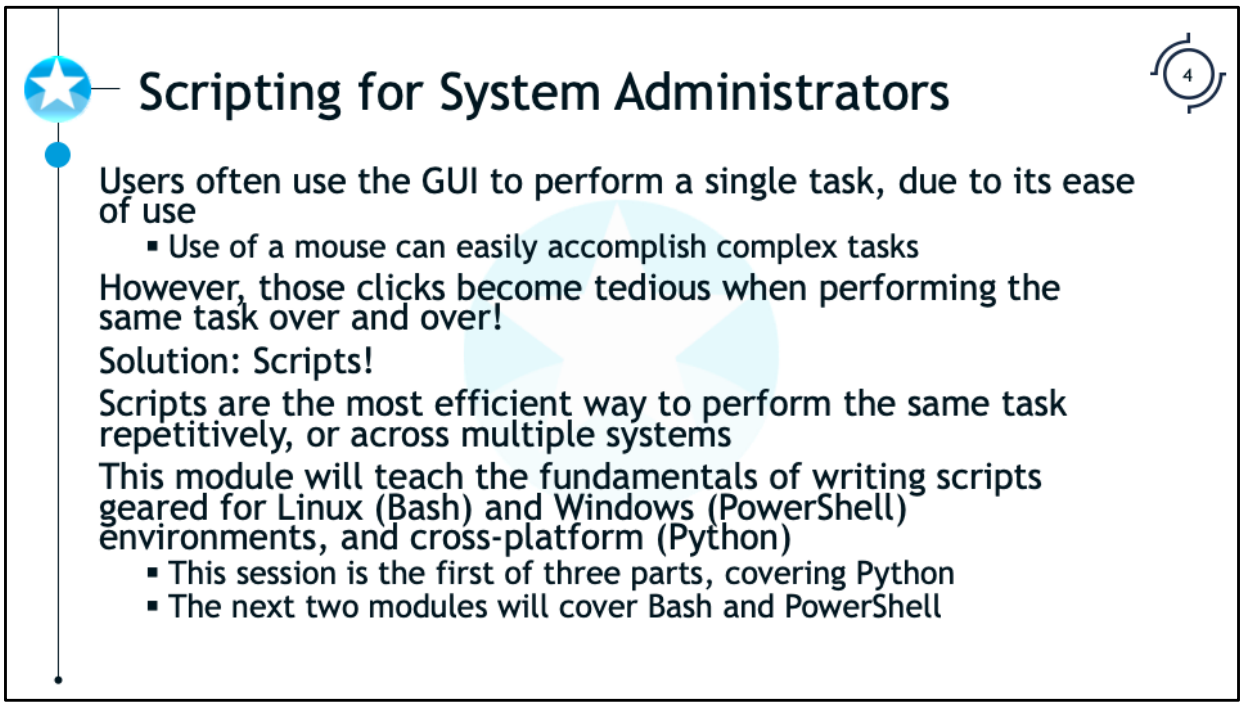
This training material was originally developed to help students, teachers, and mentors prepare for the Cyber Aces Online Competition. This module focuses on the basics of system administration and scripting. . This session is part of Module 3, System Administration. This module is split into three sections, Bash, PowerShell, and Python. In this session, we will continue our examination of Bash.

The three modules of Cyber Aces Online are Operating Systems, Networking, and System Administration.

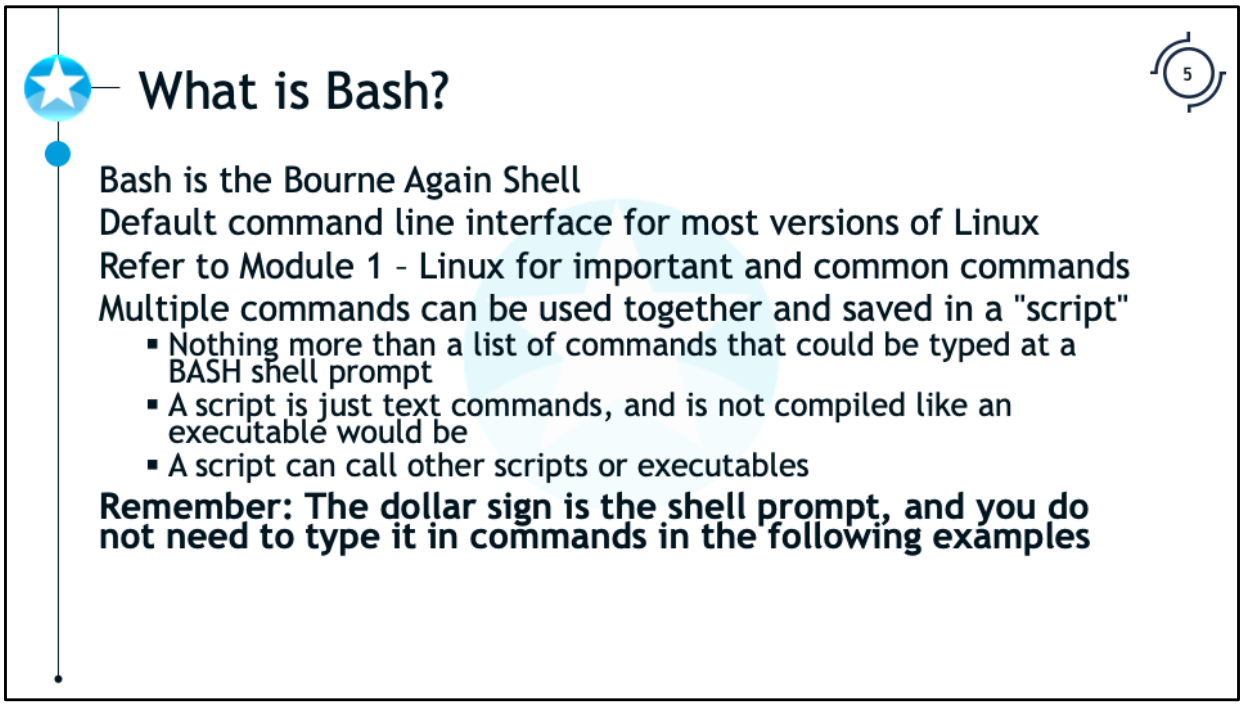
For more information about the Cyber Aces program, please visit the Cyber Aces website at https://CyberAces.org/.



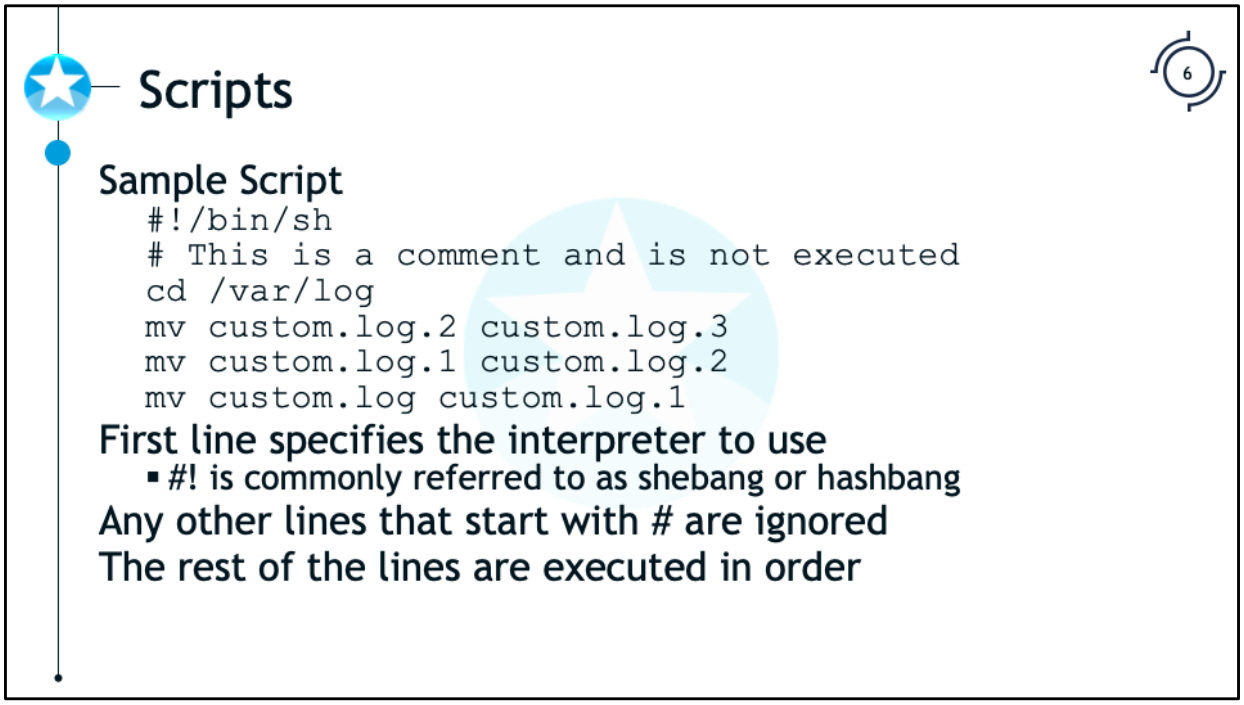
Is this section, you will be introduced to BASH and scripts and we will quickly review a few key topics from from the Linux section in Module 1.



Individuals who need to do a single task on a single machine will often use the GUI because of its ease of use. With just a few clicks of a mouse, you can easily accomplish complex tasks. However, those same clicks become tedious when the same task must be performed several times a day or performed on multiple computers. In those cases, resourceful IT professionals will often resort to developing command line scripts. A command line script is written once and can easily be run repeatedly, or be scheduled to run automatically by the computer. When you are supporting hundreds or thousands of computers, manually interacting with the programs becomes impractical and scripting is the only option. This course is intended to provide you with the tools you need to perform common administrative functions in some of the most popular scripting environments. We will examine Python, and then we will examine using GNU Bash and Microsoft PowerShell scripting from the command line to complete every day administrative functions.



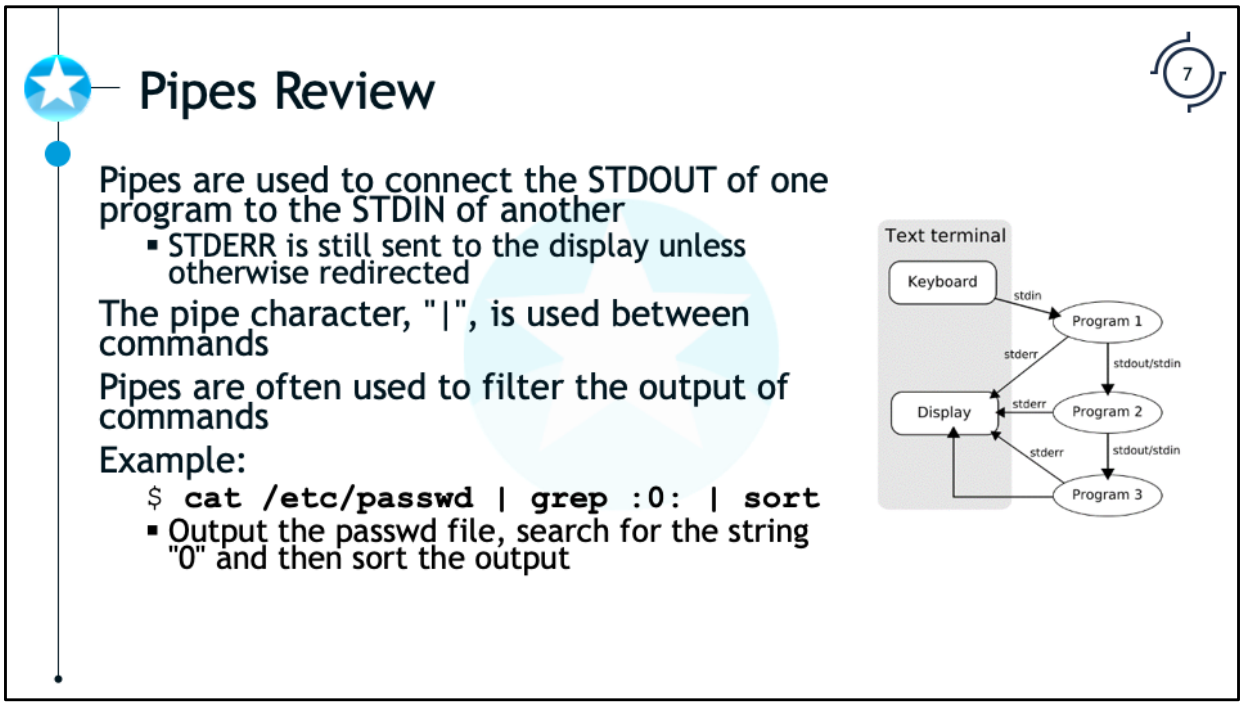
A BASH script is a script that is run through the BASH (Bourne Again Shell) Shell. In its simplest form, a BASH script is nothing more than a list of commands that would otherwise be typed interactively at a BASH shell prompt. Scripts make it easier to create complex combinations of commands and reuse the series of commands. BASH has the ability to capture, manipulate and branch execution depending on the results of those commands. BASH can read and modify files in the file system. BASH can also manipulate processes and automatically perform many routine tasks for you.



The `#!/bin/sh` is used to specify the interpreter to use. In this case, it specifies the Bourne Shell, but could have been the Bourne Again Shell (`/bin/bash`), C shell (`/bin/csh`) or any other shell or interpreter installed on the system. The code in the file must be written for the specific interpreter. For example, if `#!/usr/bin/python` was used in the first line, the remaining code would have to be understood by the Python language interpreter.

In the example script, the second line starts with a "#". The leading hash or pound sign (#) designates the line as a comment and the interpreter will ignore the line. This is commonly used to add extra information at the top of a script, such as the author, file version, and similar information. It is also used to add an additional description of what is happening in the script so it is easier for someone else to understand.

The remaining lines are executed in order. The first command changes the current directory to `/var/log`. The remaining lines rename files (or technically move them to the new name).

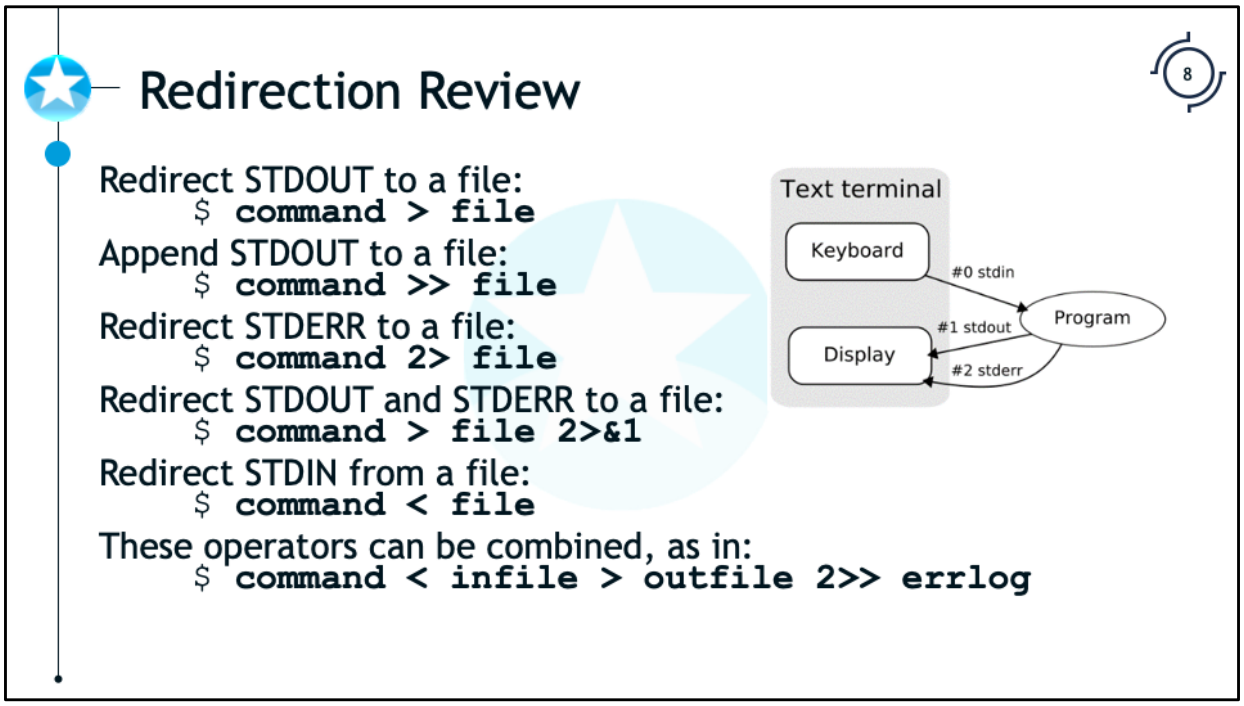


Pipes are used to connect the STDOUT of one program to the STDIN of another, creating a pipeline for data to flow through a series of programs. To use pipes, place the pipe character ("|", or shift-\) between commands. Pipes are often used to filter the output of commands, such as to search for a particular string, or to sort a set of data. The programs commonly used for these tasks (such as "grep", "sort", and "uniq") are often referred to as filters.

For example, the following command will read in the list of users on the system, search them for the string ":0:" (identifying users with UID or GID 0), and then sort them alphabetically:

```
$ cat /etc/passwd | grep :0: | sort
```

Note that STDERR is still sent to the display unless it is redirected elsewhere. This allows you to see any errors or warnings that may occur without them becoming part of the pipeline.



Redirection allows you to redirect the standard I/O streams to different locations, such as to a file or a pipe. For example, you can redirect STDIN to read data from a file instead of from the keyboard, redirect STDOUT to write to a file instead of the screen, and redirect STDERR to hide its output (such as by sending it to /dev/null, a black hole that discards any data it receives). Here are some examples:

Redirect STDIN from a file:

```
$ command < file
```

Redirect STDOUT to a file:

```
$ command > file
```

Append STDOUT to a file:

```
$ command >> file
```

Redirect STDERR to a file (note the file descriptor "2"):

```
$ command 2> file
```

Append STDOUT to a file (write STDOUT to the end of an existing file):

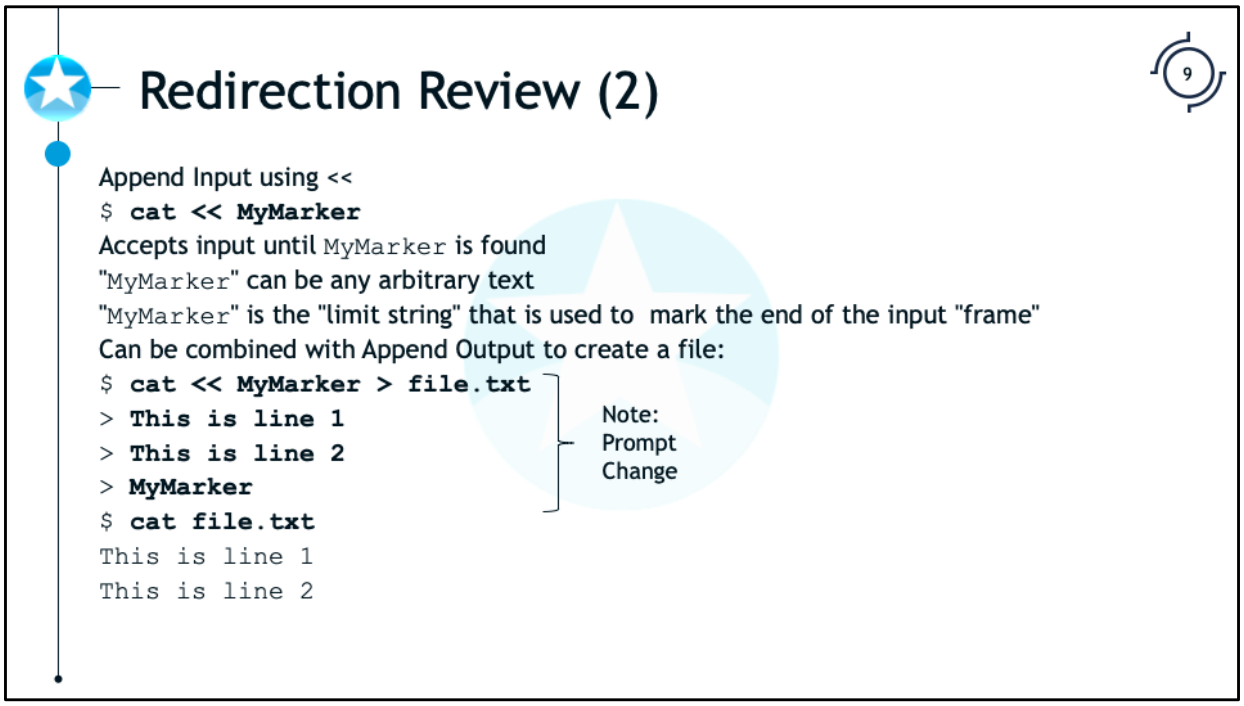
```
$ command >> file
```

Redirect STDOUT and STDERR to a file (the "2>&1" sends 2 to the same location as 1):

```
$ command > file 2>&1
```

These operators can be combined, as in:

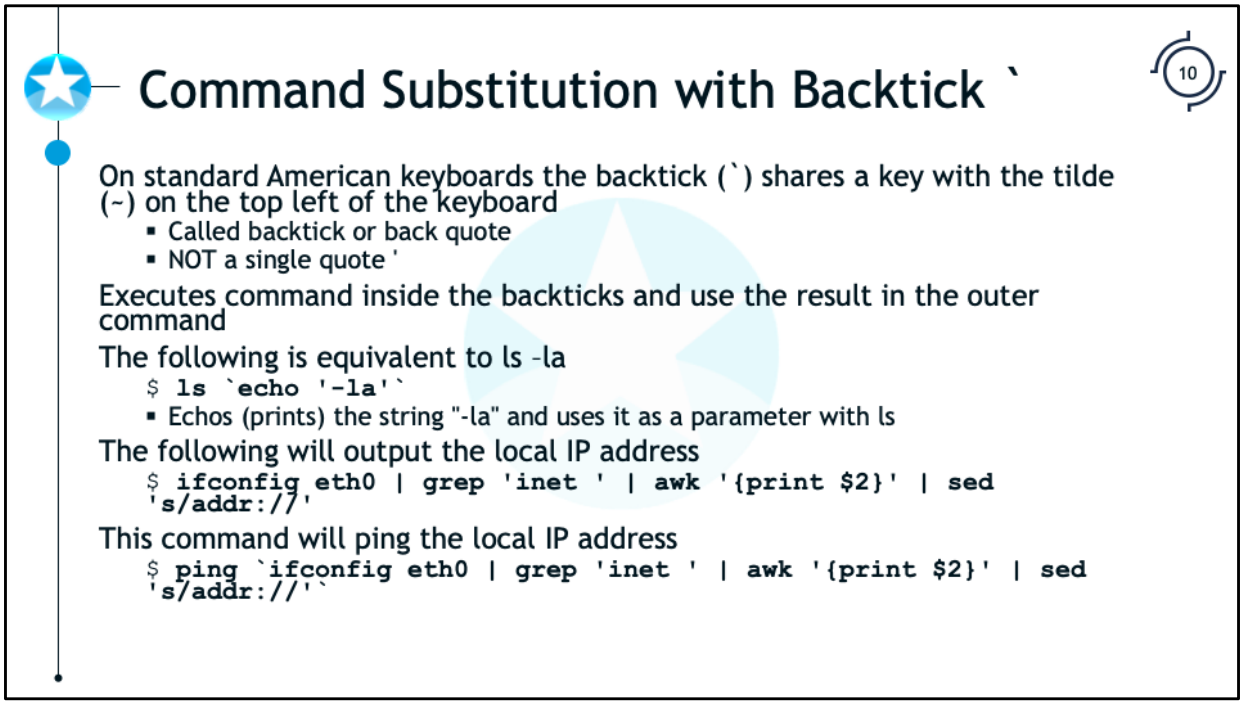
```
$ command < infile > outfile 2>> errlog
```



We have ">" and ">>" for output and "<" for input, but what about "<<"? "<<" is used for appended input. Using "<<", you can continuously append input until an End of File marker is reached. Consider the following commands typed at a BASH prompt.

```
$ cat << MyMarker > myfile.txt
> This is the first line of input.
> This is also appended to the input.
> All lines are appended until the marker is received
> MyMarker
```

This will cause "cat" to continuously accept input and print it to the screen until a line containing nothing except the end of file marker is received. In the example, we used "MyMarker" as the marker. You will often see "EOF" used as the end of file marker.



Another important syntax to learn is the use of the "`" (pronounced backtick or back quote). This is not a single quote; it is the character that shares a key with the tilde ("~"). This character tells BASH to execute the command inside the backticks and substitute the result of that command on the command line. For example, the following command is equivalent to typing "ls -la":

```
$ ls `echo '-la'`
```

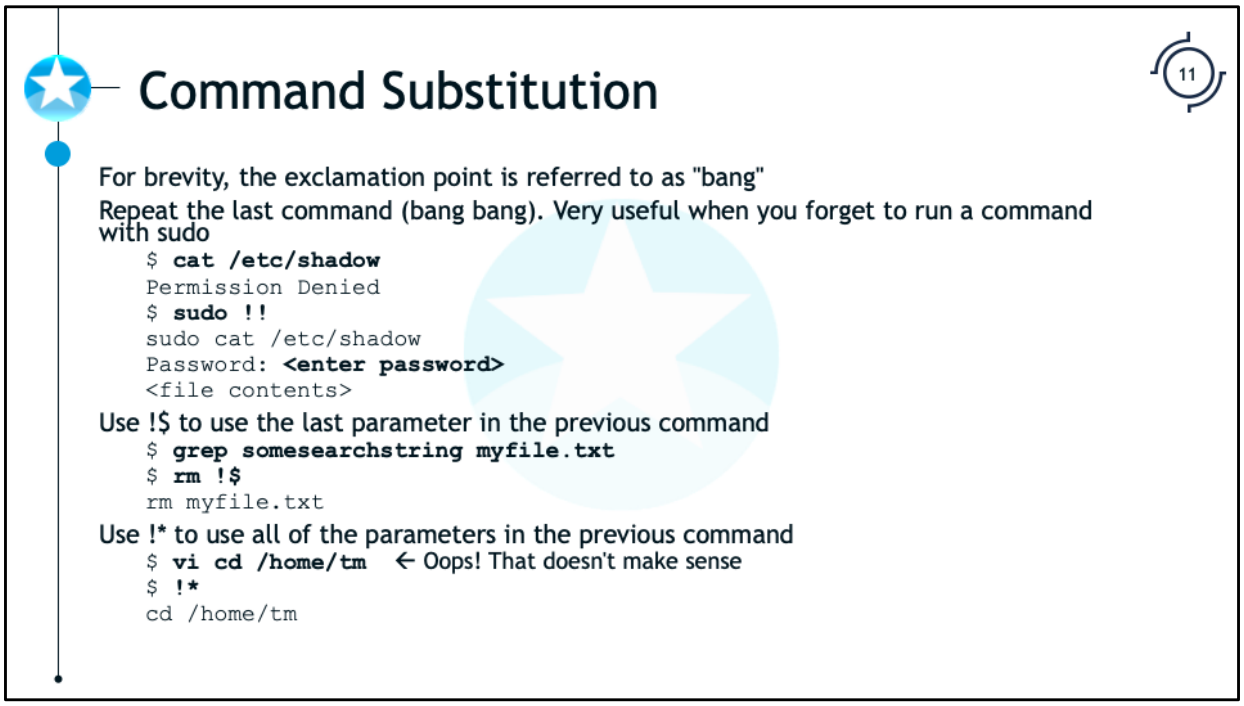
This is a very powerful feature of BASH and is used quite often in BASH programming. For example, if you want to execute the "ifconfig" command, grab your IP address from its output and then ping it, you can use command substitution to accomplish that. First, we should figure out how to grab our IP address. One option is to use the "ifconfig eth0" command, use the "grep" command to isolate the line that contains the IP address, and then use the "awk" command to grab the second field (containing the IP). Like this:

```
$ ifconfig eth0 | grep 'inet ' | awk '{print $2}' | sed
's/addr://'
```

Note: Your ethernet adapter may have a different name, such as 'ens33'. You can verify your ethernet adapter name by running "ifconfig" with no options.

The output of that string of commands is the IP address that is assigned to "eth0" (the first network adapter). Now, if you want to PING the IP address returned by that command, you could put those commands inside backticks like this:

```
$ ping `ifconfig eth0 | grep 'inet ' | awk '{print $2}' |
sed 's/addr://'`
```



The exclamation point, commonly referred to as bang for brevity, can be used to repeat the previous command or reuse parameters passed to previous commands.

It is bad practice to use the root account for day-to-day tasks, but you will sometimes need to access sensitive files or commands. When you try to access those resources without the proper permissions, you will get an access denied error. Instead of retyping the entire command (or hitting the up arrow and moving to the beginning of the line) you can simply type "sudo !!". The bang bang will be replaced with the previous command. Of course, this isn't just limited for use with sudo, but it is the most frequent use of this shortcut.

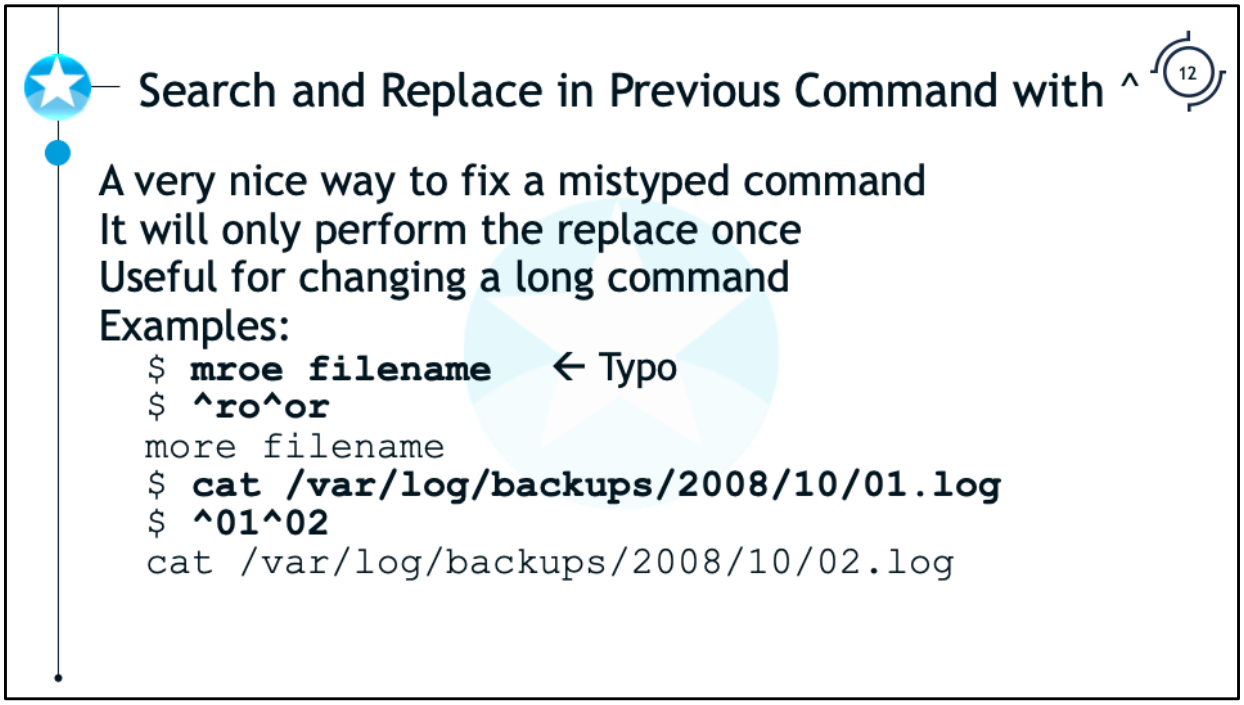
Two similar shortcuts include !$ and !*. The !$ will be replaced with the last parameter in the previous command and the !* will be replaced with all the parameter to the previous command.

Example of searching a file and then deleting it:

```
$ grep somesearchstring myfile.txt
$ rm !$
rm myfile.txt
```

Example of fixing a typo with !*

```
$ vi cd /home/tm
$ !*
cd /home/tm
```



The caret (also called hat or circumflex) can be used to rerun the previous command but replace a string within the command first. This is a very handy and efficient way to fix typos or to repeat a previous command but without a lot of retyping or editing. The syntax is:

^search^replacement

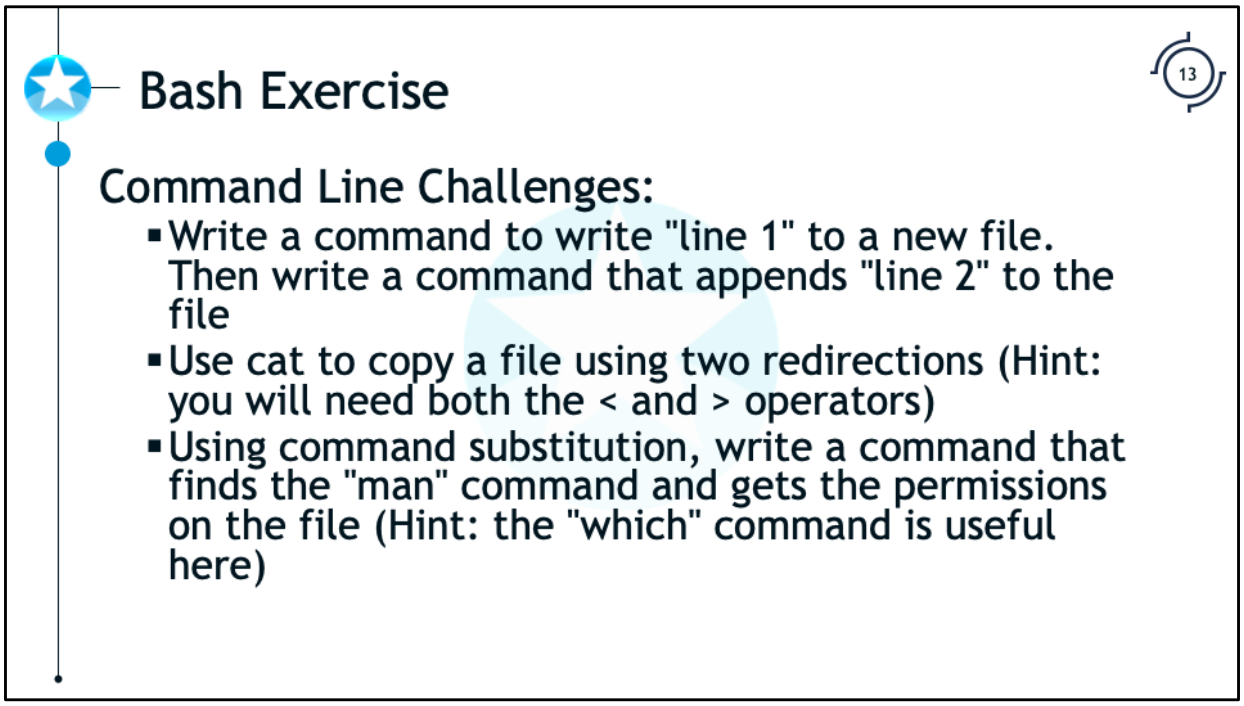
The following mistyped command can be quickly fixed without retyping (there is a misspelling of the word install).

```
# yum isntall firefox
No such command isntall
# ^sn^ns
yum install firefox
```

Notice in this case, and with the bang commands on the previous page, that the updated command is echoed on the command line for you.

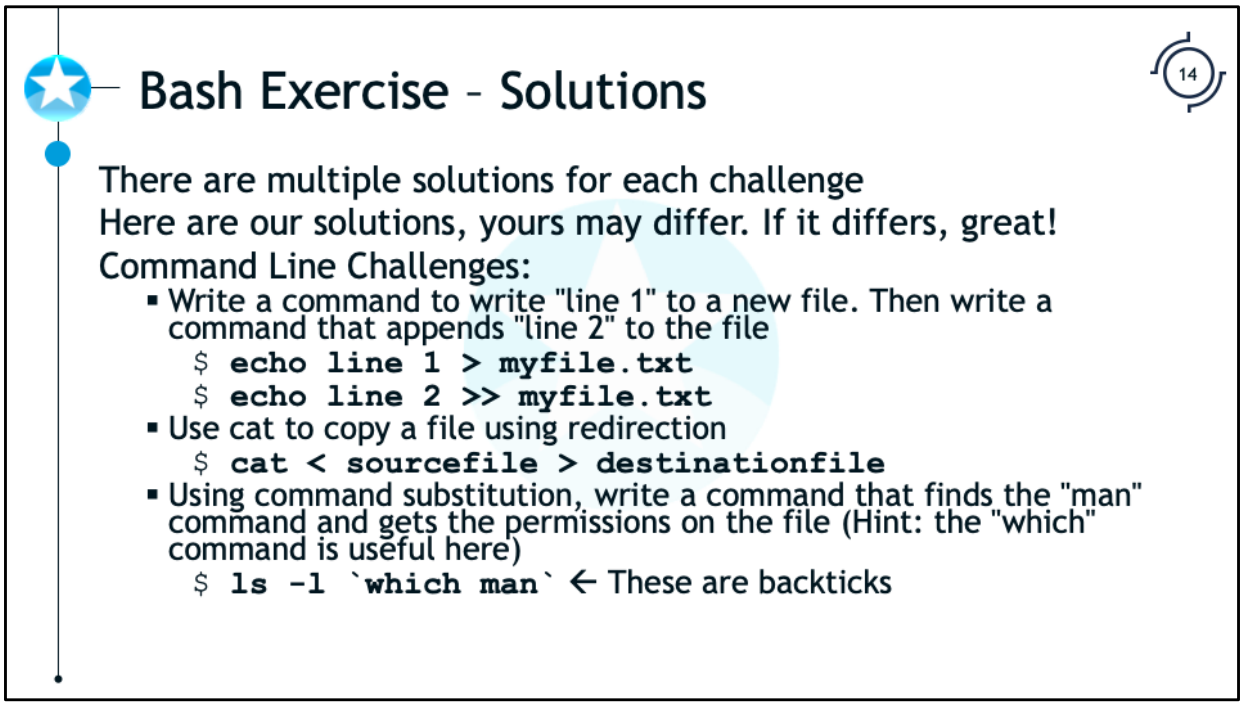
This is also very handy to modify a previous command without a lot of retyping or editing.

```
$ cat /var/log/myservice/logs/2010/10/24.php | grep error
| cut -d: -f 4
$ ^24^25
cat /var/log/myservice/logs/2010/10/25.php | grep error |
cut -d: -f 4
```



Here are three challenges for you to complete using the concepts learned in this tutorial:

1. Write a command to write "line 1" to a new file. Then write a command that appends "line 2" to the file.
2. Use cat to copy a file using redirection (Hint: you will need both the < and > operators).
3. Using command substitution, write a command that finds the "man" command and gets the permissions on the file (Hint: the "which" command is useful here).



There are multiple answers for these challenges. If you answer differs from ours, great! Here are our solutions for the challenges.

1. Write a command to write "line 1" to a new file. Then write a command that append "line 2" to the file.

   ```
   $ echo line 1 > myfile.txt
   $ echo line 2 >> myfile.txt
   ```

   The first command will create the file and overwrite the file if it already exists. The second command will append the text to the file and will not overwrite the file.

   Another more advanced solution using substitution is below. Note: The text "Line 1" would be appended to an already existing file, it would not overwrite the file.

   ```
   $ echo line 1 >> myfile.txt
   $ ^1^2
   ```

2. Use cat to copy a file using redirection (Hint: you will need both the < and > operators).

   ```
   $ cat < inputfile > outputfile
   ```

   The text in the input file is fed into cat and is sent to STDOUT. The > is then used to write the file. The following command would accomplish the same task, but it doesn't demonstrate the use of an input file.

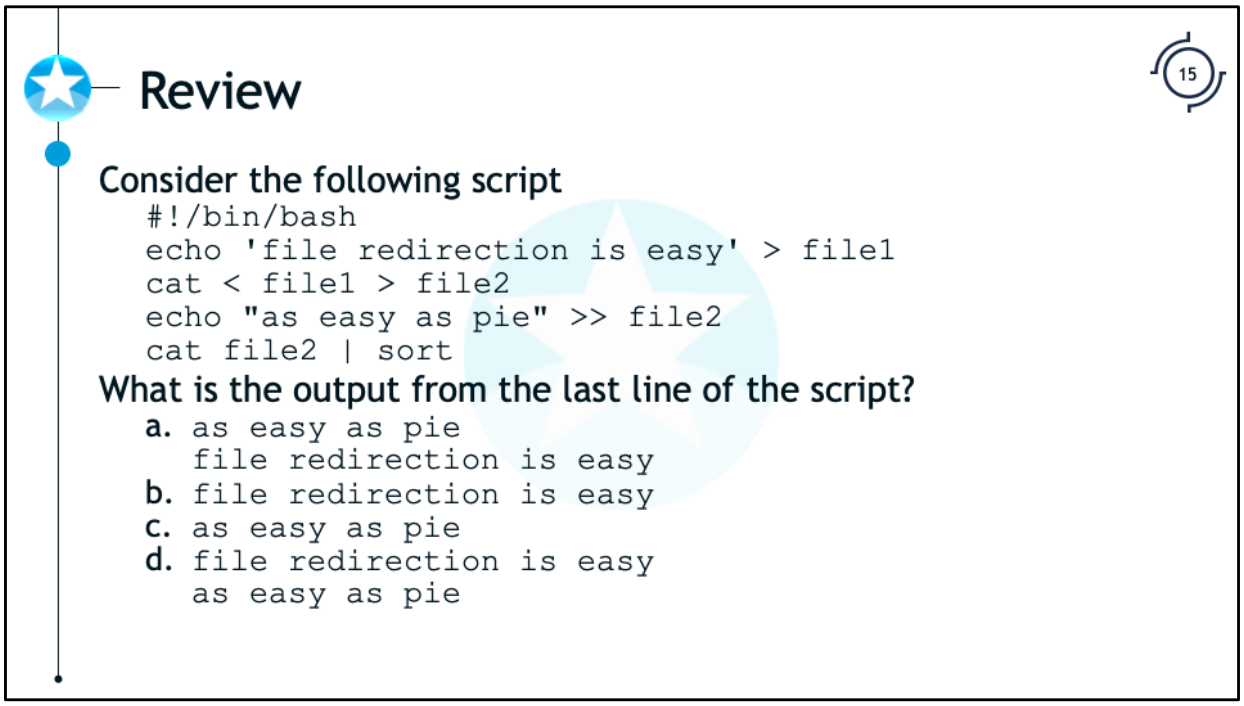
   ```
   $ cat inputfile > outputfile
   ```

3. Using command substitution, write a command that finds the "man" command and gets the permissions on the file (Hint: the "which" command is useful here).

   ```
   $ ls -l `which man`
   ```

   The -l (lowercase L) option will give us the long format which includes the

permissions.

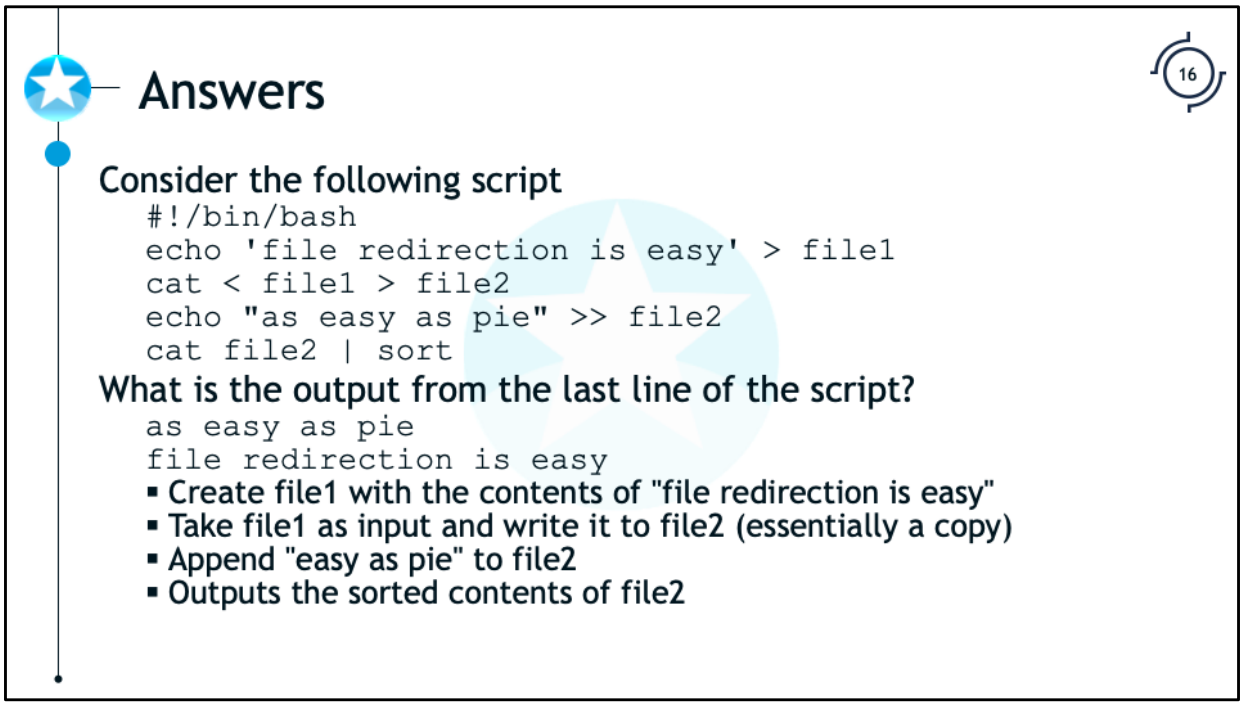


Consider the following script

```
#!/bin/bash
echo 'file redirection is easy' > file1
cat < file1 > file2
echo "as easy as pie" >> file2
cat file2 | sort
```

What is the output from the last line of the script?

a.
```
as easy as pie
file redirection is easy
```
b.
```
file redirection is easy
```
c.
```
as easy as pie
```
d.
```
file redirection is easy
as easy as pie
```



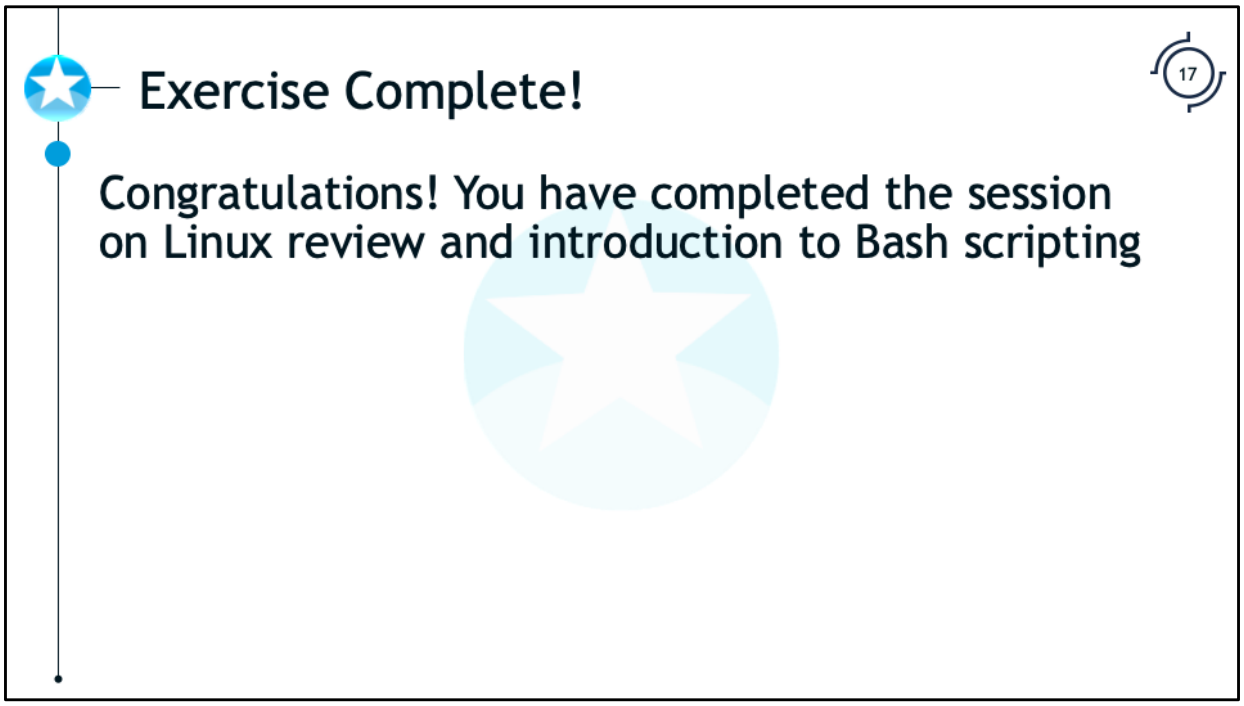
What is the output from the last line of the script?

The answer is A

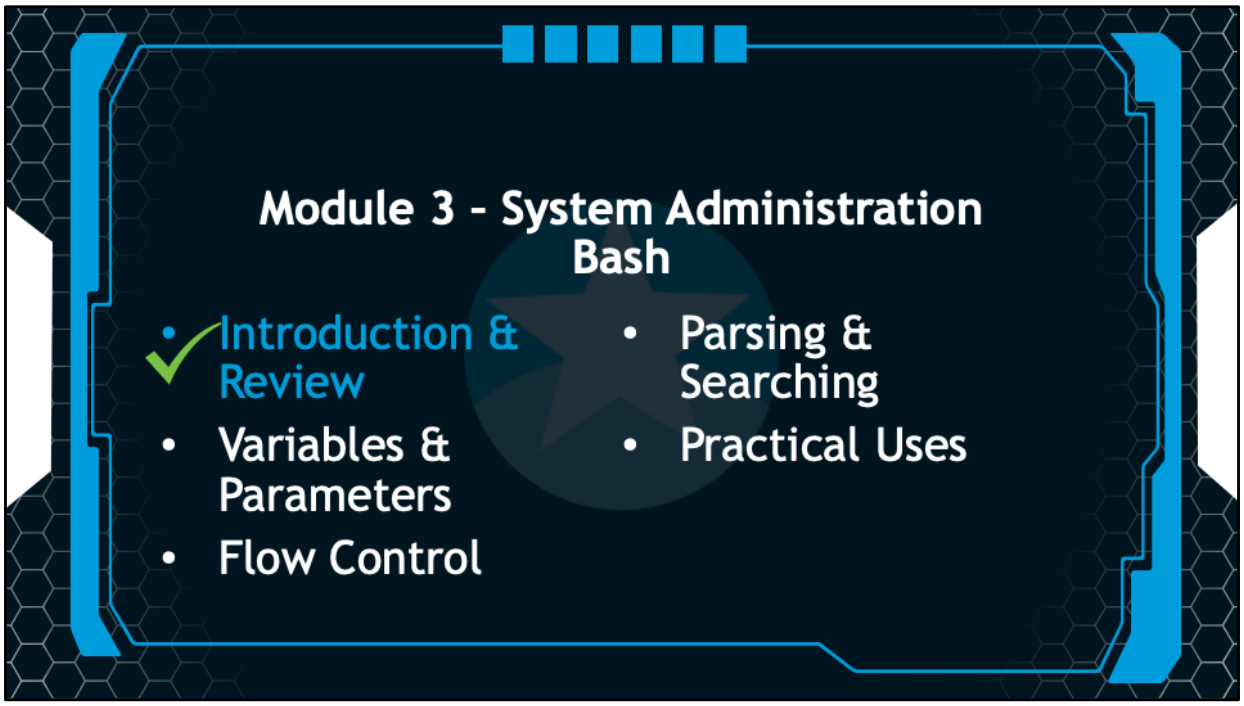
```
as easy as pie
```

```
file redirection is easy
```

- Create file1 with the contents of "file redirection is easy"
- Take file1 as input and write it to file2 (essentially a copy)
- Append "easy as pie" to file2
- Outputs the sorted contents of file2



Congratulations! You have completed the session on Linux review and introduction to Bash scripting.



In the next session, we will discuss variables and script parameters.