Examples of Bash Shell Features

Learning Objectives

After completing this reading, you will be able to:

- List examples of metacharacters
- Use quoting to specify literal or special character meanings
- Implement input and output redirection
- Apply command substitution
- Describe applications for command line arguments

Metacharacters

Metacharacters are characters having special meaning that the shell interprets as instructions.

Metacharacter	Meaning
`#`	Precedes a comment
`;`	Command separator
`*`	Filename expansion wildcard
`?`	Single character wildcard in filename expansion

Pound

The pound `#` metacharacter is used to represent comments in shell scripts or configuration files. Any text that appears after a `#` on a line is treated as a comment and is ignored by the shell.

```
#!/bin/bash
# This is a comment
echo "Hello, world!"  # This is another comment
```

Comments are useful for documenting your code or configuration files, providing context, and explaining the purpose of the code to other developers who may read it. It's a best practice to include comments in your code or configuration files wherever necessary to make them more readable and maintainable.

Semicolon ;

The semicolon `;` metacharacter is used to separate multiple commands on a single command line. When multiple commands are separated by a semicolon, they are executed sequentially in the order they appear on the command line.

```
$ echo "Hello, "; echo "world!"
Hello,
world!
```

As you can see from the example above, the output of each `echo` command is printed on separate lines and follows the same sequence in which the commands were specified.

The semicolon metacharacter is useful when you need to run multiple commands sequentially on a single command line.

Asterisk *

The asterisk `*` metacharacter is used as a wildcard character to represent any sequence of characters, including none.

```
ls *.txt
```

In this example, `*.txt` is a wildcard pattern that matches any file in the current directory with a `.txt` extension. The `ls` command lists the names of all matching files.

Question mark `?`

The question mark `?` metacharacter is used as a wildcard character to represent any single character.

```
ls file?.txt
```

In this example, `file?.txt` is a wildcard pattern that matches any file in the current directory with a name starting with `file`, followed by any single character, and ending with the `.txt` extension.

Quoting

Quoting is a mechanism that allows you to remove the special meaning of characters, spaces, or other metacharacters in a command argument or shell script. You use quoting when you want the shell to interpret characters literally.

Symbol	Meaning
`\`	Escape metacharacter interpretation
`" "`	Interpret metacharacters within string
`' '`	Escape all metacharacters within string

Backslash `\`

The backslash character is used as an escape character. It instructs the shell to preserve the literal interpretation of special characters such as space, tab, and `$`. For example, if you have a file with spaces in its name, you can use backslashes followed by a space to handle those spaces literally:

```
touch file\ with\ space.txt
```

Double quotes `" "`

When a string is enclosed in double quotes, most characters are interpreted literally, but metacharacters are interpreted according to their special meaning. For example, you can access variable values using the dollar `$` character:

```
$ echo "Hello $USER"
Hello <username>
```

Single quotes `' '`

When a string is enclosed in single quotes, all characters and metacharacters enclosed within the quotes are interpreted literally. Single quotes alter the above example to produce the following output:

```
$ echo 'Hello $USER'
```

```
Hello $USER
```

Notice that instead of printing the value of `$USER`, single quotes cause the terminal to print the string `"$USER"`.

Input/Output redirection

Symbol	Meaning
`>`	Redirect output to file, overwrite
`>>`	Redirect output to file, append
`2>`	Redirect standard error to file, overwrite
`2>>`	Redirect standard error to file, append
`<`	Redirect file contents to standard input

Input/output (IO) redirection is the process of directing the flow of data between a program and its input/output sources.

By default, a program reads input from *standard input*, the keyboard, and writes output to *standard output*, the terminal. However, using IO redirection, you can redirect a program's input or output to or from a file or another program.

Redirect output `>`

This symbol is used to redirect the standard output of a command to a specified file.

`ls > files.txt` will create a file called `files.txt` if it doesn't exist, and write the output of the `ls` command to it.

Warning: When the file already exists, the output overwrites all of the file's contents!

Redirect and append output `>>`

This notation is used to redirect and append the output of a command to the end of a file. For example,

`ls >> files.txt` appends the output of the `ls` command to the end of file `files.txt`, and preserves any content that already existed in the file.

Redirect standard output `2>`

This notation is used to redirect the standard error output of a command to a file. For example, if you run the ls command on a non-existing directory as follows,

`ls non-existent-directory 2> error.txt` the shell will create a file called `error.txt` if it doesn't exist, and redirect the error output of the `ls` command to the file.

Warning: When the file already exists, the error message overwrites all of the file's contents!

Append standard error `2>>`

This symbol redirects the standard error output of a command and appends the error message to the end of a file without overwriting its contents.

`ls non-existent-directory 2>> error.txt` will append the error output of the `ls` command to the end of the `error.txt` file.

Redirect input `<`

This symbol is used to redirect the standard input of a command from a file or another command. For example,

`sort < data.txt` will `sort` the contents of the `data.txt` file.

Command Substitution

Command substitution allows you to run command and use its output as a component of another command's argument. Command substitution is denoted by enclosing a command in either backticks (`**command**`) or using the `$()` syntax. When the encapsulate command is executed, its output is substituted in place, and it can be used as an argument within another command. This is particularly useful for automating tasks that require the use of a command's output as input for another command.

For example, you could store the path to your current directory in a variable by applying command substitution on the `pwd` command, then move to another directory, and finally return to your original directory by invoking the `cd` command on the variable you stored, as follows:

```
$ here=$(pwd)
$ cd path_to_some_other_directory
$ cd $here
```

Command Line Arguments

Command line arguments are additional inputs that can be passed to a program when the program is run from a command line interface. These arguments are specified after the name of the program, and they can be used to modify the behavior of the program, provide input data, or provide output locations. Command line arguments are used to pass arguments to a shell script.

For example, the following command provides two arguments, arg1, and arg2, that can be accessed from within your Bash script:

```
$ ./MyBashScript.sh arg1 arg2
```

Summary

In this reading, you learned that:

- Metacharacters such as `#`, `;`, `*`, and `?` are characters that the shell interprets with special meanings
- Quoting allows you to ensure any special characters, spaces, or other metacharacters are interpreted literally by the shell
- Input/output redirection redirects a program's input or output to/from a file
- Command substitution allows you to use the output of a command as an argument for another command
- Command line arguments can be used to pass information to a shell script