### **What is a Here Document?**

A **Here Document** is a method in shell scripting that allows you to pass a block of text to a command as input. It is particularly useful for feeding multiple lines of input to commands like cat, grep, or sed without having to manually echo each line.

### **Syntax**

The basic syntax for a Here Document is:

command <<EOF

# multiple lines of input

EOF

* **<<EOF**: The << operator followed by a delimiter (EOF is commonly used but you can use any string).
* **Input Block**: The lines between <<EOF and EOF are passed as input to the command.
* **EOF**: The end marker, indicating the end of the Here Document. The marker must be the same as the one used after <<.

### **Example 1: Using cat with a Here Document**

#!/bin/bash

# Use a here document to pass multiple lines to cat

cat <<EOF

This is a sample text.

It contains multiple lines.

These lines will be displayed by cat.

EOF

**Output:**

This is a sample text.

It contains multiple lines.

These lines will be displayed by cat.

* In this example, cat reads the lines between <<EOF and EOF and displays them on the screen.

### **Example 2: Writing to a File Using cat and Here Document**

#!/bin/bash

# Write multiple lines to a file using a here document

cat <<EOF > output.txt

This text will be written to output.txt.

The file will contain multiple lines of text.

EOF

* **> output.txt**: Redirects the output of cat to output.txt, creating the file if it doesn't exist, or overwriting it if it does.

**Content of output.txt:**

This text will be written to output.txt.

The file will contain multiple lines of text.

### **Example 3: Appending to a File Using cat and Here Document**

#!/bin/bash

# Append multiple lines to an existing file using a here document

cat <<EOF >> output.txt

This text will be appended to output.txt.

EOF

* **>> output.txt**: Appends the output to output.txt instead of overwriting it.

**Updated Content of output.txt:**

This text will be written to output.txt.

The file will contain multiple lines of text.

This text will be appended to output.txt.

### **Example 4: Using Variables in a Here Document**

#!/bin/bash

# Declare a variable

name="John Doe"

# Use the variable in a here document

cat <<EOF

Hello, $name.

Welcome to the Here Document tutorial.

EOF

**Output:**

Hello, John Doe.

Welcome to the Here Document tutorial.

* Variables inside a Here Document are expanded, meaning their values are inserted into the text.

### **Example 5: Preventing Variable Expansion**

If you want to prevent variable expansion (i.e., treat variables as plain text), you can quote the delimiter:

#!/bin/bash

# Prevent variable expansion

cat <<'EOF'

Hello, $name.

Variables will not be expanded here.

EOF

**Output:**

Hello, $name.

Variables will not be expanded here.

* **<<'EOF'**: Quoting the delimiter prevents the shell from interpreting variables, so they appear as is.

**Logger command**

**The logger command in shell scripting is used to write messages to the system log (syslog). It's particularly useful for logging important information, warnings, or errors from scripts to a centralized logging system, where they can be monitored and analyzed.**

### **Basic Syntax**

**logger [options] message**

* **message: The text or message you want to log.**
* **options: Additional options to specify the priority, tag, etc.**

### **Common Options**

* **-p priority: Specifies the priority of the log message. The format is facility.level, e.g., user.info.**
* **-t tag: Specifies a tag to identify the source of the log message.**
* **-i: Logs the process ID (PID) with the message.**
* **-f file: Logs the contents of the specified file.**

**​​**

**#!/bin/bash**

**# Run the df command and redirect output to /tmp/dfh**

**df -kh > /tmp/dfh**

**# Check the exit status of the df command**

**if [ $? -eq 0 ]; then**

**# If the command was successful, log a success message**

**logger -t myscript "df command executed successfully." -f /var/log/messages**

**else**

**# If the command failed, log a failure message**

**logger -t myscript "df command failed to execute." -f /var/log/messages**

**fi**

**Debugging shell scripts**

### **Debugging with bash -x**

Run your script with the -x option to see the commands being executed.

**Example:**  
bash -x script.sh

* + **Output**: This will run script.sh with tracing enabled, showing each command and its result.

### **Using set Command for Debugging**

The set command with different options helps you identify errors, trace the execution flow, and see which commands are executed.

**set -x (or set -o xtrace)**: Enables a mode of the shell where all executed commands are printed to the terminal before execution.  
#!/bin/bash

set -x # Start tracing

echo "This is a test"

var=$(ls non\_existing\_file) # This will fail

echo $var

set +x # Stop tracing

* + **Output**: The script will print each command and its result to the terminal, helping you trace the flow.

**set -e (or set -o errexit)**: Causes the script to exit immediately if any command returns a non-zero status (i.e., an error).  
  
#!/bin/bash

set -e # Exit on error

echo "This is a test"

var=$(ls non\_existing\_file) # This will cause the script to exit

echo $var

Echo “hello world”

* + **Output**: The script will stop execution at the point where the command fails.

**Running a Script in the Background**

### **Running a Script in the Background with &**

The simplest way to run a script in the background is to append & to the end of the command.

#### **Example**

./my\_script.sh &

* **Explanation**: This command starts my\_script.sh in the background, allowing you to continue using the terminal. The script’s process ID (PID) will be displayed in the terminal, so you can track or manage it if needed.

### **Running a Script in the Background with nohup**

If you want your script to continue running even after you close the terminal, you can use nohup (no hang up).

#### **Example**

nohup ./my\_script.sh &

* **Explanation**: nohup prevents the script from being terminated when you log out or close the terminal. The output will be redirected to a file named nohup.out unless you specify a different output file.

### **Redirecting Output While Running in the Background**

To avoid cluttering the terminal with output from your script, you can redirect both standard output and standard error to a file.

#### **Example**

./my\_script.sh > /tmp/my\_script.log 2>&1 &

* **Explanation**: This command runs my\_script.sh in the background and redirects all output to /tmp/my\_script.log. The 2>&1 part redirects the error output to the same file as the standard output.

Both & and nohup are used to run commands in the background in a Unix-like environment, but they serve different purposes and have distinct behaviors. Here’s a breakdown of their differences:

### **1. & (Ampersand)**

* **Purpose**: The & is used to run a command in the background, allowing you to continue using the terminal for other tasks.
* **Behavior**:
  + When you append & to a command, it starts that command in the background.
  + The shell immediately returns control to you, displaying the process ID (PID) of the background job.
  + The command will keep running as long as the terminal session is active.
  + If you close the terminal or log out, the background process started with & will be terminated unless it has been specifically detached.

**Example**:  
  
./my\_script.sh &

* + **Explanation**: my\_script.sh runs in the background, and the terminal remains available for other commands. However, if you close the terminal, the script will stop.

### **2. nohup (No Hangup)**

* **Purpose**: nohup is used to run a command that continues to run even after the terminal or session is closed.
* **Behavior**:
  + When you use nohup, the command is immune to hangups (HUP signals) and will continue running even after you log out or close the terminal.
  + By default, nohup redirects the command's output to a file called nohup.out unless you specify a different file.
  + nohup can be used with or without &. Using nohup without & will run the command in the foreground, whereas combining nohup with & runs the command in the background.

**Example**:  
  
nohup ./my\_script.sh &

* + **Explanation**: This runs my\_script.sh in the background and ensures it continues to run even if you close the terminal or log out. The output is saved in nohup.out unless redirected elsewhere.

seq command

The seq command in Unix/Linux is used to generate a sequence of numbers, printing each number on a new line. It is useful in various scripting scenarios where you need to create a sequence of numbers or iterate over a range of values.

### **Basic Syntax**

seq [OPTION]... LAST

seq [OPTION]... FIRST LAST

seq [OPTION]... FIRST INCREMENT LAST

### **Examples**

**Generate a Sequence from 1 to a Given Number**bash  
Copy code  
seq 5

**Output**:  
  
1

2

3

4

5

* + **Explanation**: This command generates a sequence of numbers from 1 to 5.

**Generate a Sequence with a Specified Start and End**  
seq 3 7

**Output**:  
  
3

4

5

6

7

* + **Explanation**: This generates numbers from 3 to 7.

**Generate a Sequence with a Specified Increment**  
seq 1 2 9

**Output**:  
  
1

3

5

7

9

* + **Explanation**: This generates numbers from 1 to 9, incrementing by 2.

**Generating a List of IP Addresses**:

for i in $(seq 1 5); do

echo "192.168.1.$i"

done

**Difference between $\* and $@**

The difference between $\* and $@ in shell scripting lies in how they handle positional parameters (arguments passed to a script or a function).

### **$\* (All Positional Parameters as a Single String)**

* When you use $\*, all the positional parameters are treated as a single string.
* If you use $\* within double quotes ("$\*"), the parameters are concatenated into a single string with the first character of the IFS (Internal Field Separator) between them. By default, this separator is a space.

### **$@ (All Positional Parameters as Separate Strings)**

* When you use $@, all the positional parameters are treated as separate strings.
* If you use $@ within double quotes ("$@"), each positional parameter remains a separate string. This is usually the preferred way to pass all arguments to another command or script, as it preserves the exact nature of the arguments.

#!/bin/bash

# Greet the user using positional parameters $1, $2, and $3

echo "Hello, $1 $2! $3"

# Loop through the arguments using $\*

echo "Looping through arguments using \"\$\*\":"

for arg in "$\*"; do

echo "$arg"

done

# Loop through the arguments using $@

echo "Looping through arguments using \"\$@\":"

for arg in "$@"; do

echo "$arg"

done

### **Explanation:**

1. **Greeting:** The script starts by greeting the user with the values of $1, $2, and $3, which represent the first, second, and third positional parameters passed to the script.
2. **Loop using $\*:**
   * When using "$\*", all the arguments are treated as a single string, and the loop will iterate only once over this combined string.
3. **Loop using $@:**
   * When using "$@", each argument is treated as a separate string, so the loop will iterate over each argument individually.

### **Example Usage:**

Save this script as script.sh, then run it with arguments:

$ ./script.sh John Doe "Welcome to the team"

### **Example Output:**

Hello, John Doe! Welcome to the team

Looping through arguments using "$\*":

John Doe Welcome to the team

Looping through arguments using "$@":

John

Doe

Welcome to the team

**Difference between single quotes and Double quotes**

**​​Single Quotes (' ')**

* **Literal Interpretation: Text enclosed in single quotes is treated literally. This means that everything inside single quotes is preserved exactly as it is, with no interpretation of variables, commands, or escape sequences.**
* **No Variable Expansion: Variables inside single quotes are not expanded; they are treated as plain text.**
* **No Command Substitution: Commands inside single quotes are not executed; they are treated as plain text.**
* **No Escape Characters: Backslashes (\) are treated as ordinary characters inside single quotes.**

**Example:**

**#!/bin/bash**

**name="John"**

**echo 'Hello, $name!'**

**Output:**

**Hello, $name!**

**Here, $name is not expanded to "John"; it is treated as a literal string.**

### **Double Quotes (" ")**

* **Variable Expansion: Text enclosed in double quotes is subject to variable expansion, meaning that variables inside double quotes are replaced with their values.**
* **Command Substitution: Commands inside double quotes are executed, and their output is included in the string.**
* **Escape Sequences: Some escape sequences like \n (newline) and \t (tab) are interpreted inside double quotes. Backslashes can be used to escape certain characters (e.g., \" to include a literal double quote).**

**Example:**

**#!/bin/bash**

**name="John"**

**echo "Hello, $name!"**

**Output:**

**Hello, John!**

**Here, $name is expanded to "John" because double quotes allow variable expansion.**

**Run command to remote server via ssh**

**#!/bin/bash**

**# Define variables for the remote server**

**REMOTE\_USER="john"**

**REMOTE\_HOST="192.168.1.100"**

**REMOTE\_COMMAND="ls -l /home/john"**

**# Use SSH to connect to the remote server, run the command, and then exit**

**ssh ${REMOTE\_USER}@${REMOTE\_HOST} "${REMOTE\_COMMAND}"**