Shell Script

16 February 2025 10:24 AM

Input in ShellScript: Reading User Input

In Bash scripting, the read command is a versatile tool that allows you to capture input from users. This can include single variables, multiple inputs, passwords, or even advanced options for flexibility.

Basic Syntax

The general syntax of the read command is:

```
read [options] [variable_names]
```

- **options**: Flags to modify the behavior of the read command (e.g., silent input).
- **variable_names**: The names of the variables where input data will be stored.

Examples

Reading a Single Value

Capture a single input value and display it back to the user:

```
#!/bin/bash
read name
echo "Hello, ${name}!"
```

Reading Multiple Values

You can read multiple variables at once:

```
#!/bin/bash
read name age
echo "Hello ${name}, you are ${age} years old."
```

Using a Prompt Message

Add a custom prompt message to make the input process more user-friendly:

```
#!/bin/bash
read -p "Please enter your name: " name
read -p "Please enter your age: " age
echo "Hello ${name}, you are ${age} years old."
```

Reading Secret Input

To securely capture sensitive data like passwords, use the -s option:

```
#!/bin/bash
read -p "Please enter your password: " -s password
echo
echo "Your password is stored securely."
```

Combining Prompt and Secret Input

```
#!/bin/bash
read -p "Enter your username: " username
read -p "Enter your password: " -s password
```

```
echo
echo "Welcome, ${username}!"
```

Advanced Usage

Specifying a Timeout

The -t option allows you to specify a timeout (in seconds) for user input:

```
#!/bin/bash
read -t 10 -p "Enter your name within 10 seconds: " name
echo "Hello, ${name}!"
```

Reading Default Values

Set a default value for a variable if no input is provided:

```
#!/bin/bash
read -p "Enter your name [Default: Gaurav]: " name
name=${name:-Gaurav}
echo "Hello, ${name}!"
```

Reading Input Into an Array

Use the -a option to read inputs into an array:

```
#!/bin/bash
read -p "Enter multiple values: " -a values
echo "You entered: ${values[@]}"
```

Handling Special Characters

Escape special characters during input using \ or quotes:

```
#!/bin/bash
read -p "Enter a command with special characters: " command
echo "You entered: ${command}"
```

Example Script

Here's a full script demonstrating multiple features of the read command:

```
#!/bin/bash
# Demo: Read Command
# Read single input
read -p "Enter your name: " name
# Read multiple inputs
read -p "Enter your age and city: " age city
# Read password securely
read -p "Enter your password: " -s password
echo
# Display results
echo "Name: ${name}"
echo "Age: ${age}, City: ${city}"
echo "Password: (hidden)"
```

Sample Output

When you run the script above, you'll see:

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./read_script.sh
Enter your name: Gaurav
Enter your age and city: 30 New Delhi
Enter your password:
Name: Gaurav
Age: 30, City: New Delhi
Password: (hidden)
```

Best Practices

1. **Always validate user input** to avoid unexpected behavior or security vulnerabilities.
2. **Use prompt messages** to guide users.
3. **Secure sensitive input** (e.g., passwords) with the `-s` option.
4. **Provide defaults** to ensure smooth script execution in case of empty input.
5. **Use arrays** for structured data input.

Command-Line Arguments in Shell Script

Command-line arguments are a way to pass input directly during the execution of a shell script, eliminating the need to hardcode variables or read them dynamically during runtime. These arguments are captured and processed using **positional parameters** and **special variables**.

What are Positional Parameters?

Positional parameters are variables that store command-line arguments passed to a script. These are represented by `$1`, `$2`, `$3`, and so on, where `$1` is the first argument, `$2` is the second, and so forth.

Syntax

```
#!/bin/bash
variable=${1}
```

Here:

- `${1}` represents the first command-line argument.
- You can define multiple positional parameters similarly, such as `${2}`, `${3}`, etc.

Example 1: Basic Usage

```
#!/bin/bash
name=${1}
age=${2}
echo "Hello, my name is ${name} and my age is ${age}."
```

Execution

1. Grant execute permission to the script:
   ```
   chmod +x commandlineargs.sh
   ```
2. Run the script with arguments:
   ```
   ./commandlineargs.sh Gaurav 30
   ```

Output

```
Hello, my name is Gaurav and my age is 30.
```

Example 2: Displaying All Arguments

The following script demonstrates how to access all arguments and special variables:

```
#!/bin/bash
name=${1}
age=${2}
# Display all positional parameters
echo "Argument 1: ${1}"
echo "Argument 2: ${2}"
# Special Variables
echo "Script Name: ${0}"  # Name of the script
echo "Total Arguments: $#"  # Total number of arguments
echo "All Arguments (quoted): $*"  # All arguments as a single string
echo "All Arguments (individual): $@"
```

Special Variables

Variable	Description	Example
${0}	The name of the script itself.	./commandlineargs.sh
${1}, ${2} ...	The positional parameters corresponding to the arguments passed to the script.	${1} -> Gaurav, ${2} -> 30
$#	Total number of arguments passed to the script.	2
$*	All the arguments passed as a single string, quoted.	"Gaurav 30"
$@	All the arguments passed as individual words, retaining their separation.	"Gaurav" "30"
$?	Exit status of the last executed command.	0 for success, non-zero for failure.
$$	Process ID of the current shell script.	e.g., 8723
$PPID	Parent process ID of the current shell script.	e.g., 8701

Advanced Examples

Looping Through Arguments

```
#!/bin/bash
for arg in "$@"; do
   echo "Argument: ${arg}"
done
```

Execution:

```
./loopargs.sh Gaurav 30 Developer
```

Output:

```
Argument: Gaurav
Argument: 30
Argument: Developer
```

Using Total Arguments in Logic

```
#!/bin/bash
if [ $# -lt 2 ]; then
   echo "Error: At least 2 arguments are required."
   exit 1
fi
echo "You provided $# arguments."
```

Execution:
```
./countargs.sh Gaurav
```
Output:
```
Error: At least 2 arguments are required.
```

Combining Arguments
```
#!/bin/bash
combined="$*"
echo "Combined Arguments: ${combined}"
```
Execution:
```
./combineargs.sh Hello World Bash
```
Output:
```
Combined Arguments: Hello World Bash
```

Handling Special Characters in Arguments
```
#!/bin/bash
echo "First Argument: ${1}"
```
Execution:
```
./specialchar.sh "Hello & Welcome"
```
Output:
```
First Argument: Hello & Welcome
```
Best Practices
1. **Validate Input**: Ensure the required number of arguments are passed before processing.
2. **Quote Variables**: Always quote variables to handle spaces or special characters.
3. **Use Default Values**: Provide defaults for missing arguments using parameter expansion: `${1:-default_value}`.

Assign a Command's Output to a Variable in Shell Script

In Shell scripting, assigning the output of a command to a variable is a common task. You can do this using different methods depending on the syntax you prefer.

Methods to Assign Command Output
1. Using Backticks ( )
Backticks enclose the command whose output you want to capture. However, this method is outdated and less preferred in modern scripting.
```
VARIABLE_NAME=`command_here`
```
2. Using Brackets (Recommended Method)
Brackets offer a more modern and readable way to capture command output. It also supports nesting without confusion.

```
VARIABLE_NAME=$(command_here)
```

Examples

Example 1: Capture Current Directory

```
#!/bin/bash
# Capturing the current working directory
CURRENT_WORKING_DIR=$(pwd)
VARIABLE_SECOND_METHOD=`pwd`
echo "Using brackets: ${CURRENT_WORKING_DIR}"
echo "Using backticks: ${VARIABLE_SECOND_METHOD}"
```

Output:

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./assign-command-output.sh
Using brackets: /home/kali/shellscript-youtube
Using backticks: /home/kali/shellscript-youtube
```

Example 2: Capture Date and Time

```
#!/bin/bash
# Capturing the current date and time
DATE_TIME=$(date +"%D %T")
echo "Current Date and Time: ${DATE_TIME}"
```

Output:

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./assign-command-output.sh
Current Date and Time: 10/17/23 08:45:00
```

Real-life Use Cases

Use Case 1: Disk Usage Check

```
#!/bin/bash
# Capture disk usage percentage for root directory
DISK_USAGE=$(df -h / | grep '/' | awk '{print $5}')
echo "Disk Usage for /: ${DISK_USAGE}"
```

Output:

```
Disk Usage for /: 28%
```

Use Case 2: Count Files in Directory

```
#!/bin/bash
# Count the number of files in the current directory
FILE_COUNT=$(ls -1 | wc -l)
echo "Number of Files in Current Directory: ${FILE_COUNT}"
```

Output:

```
Number of Files in Current Directory: 15
```

Use Case 3: Fetch Logged-in User

```
#!/bin/bash
# Get the current logged-in user
USER_NAME=$(whoami)
echo "Logged-in User: ${USER_NAME}"
```

Output:

```
Logged-in User: gaurav
```

Why Prefer Brackets Over Backticks?

1. **Readability:** Brackets are visually clearer and easier to understand.
2. **Nesting Support:** Brackets allow nested commands without ambiguity.
3. **Modern Syntax:** Encouraged in modern scripting for consistency and maintainability.

Best Practices

- **Use descriptive variable names:** Makes scripts easier to understand.
- **Always quote variables:** Prevents issues with spaces or special characters in outputs.
- **Avoid backticks:** Stick to `$(...)` for better readability and less error-prone code.
- **Test thoroughly:** Run scripts in various environments to ensure compatibility.

How to Create a Read-Only Variable in Shell Script

In Shell scripting, you can create variables that cannot be modified or unset once declared. This is achieved using the `readonly` command. Such variables are particularly useful for defining constants or critical values that should remain unchanged throughout the execution of a script.

Syntax

To declare a variable as read-only, use the following syntax:

```
readonly VARIABLE_NAME
```

Examples

1. Basic Read-Only Variable

```
#!/bin/bash
# Declare a variable
name="Gaurav"
# Make it read-only
readonly name
# Print the variable
echo "Name: ${name}"
# Attempt to modify the variable
name="Sharma"  # This will cause an error
# Attempt to unset the variable
unset name  # This will also cause an error
```

Output:

```
Name: Gaurav
./script.sh: line 8: name: readonly variable
```

```
./script.sh: line 11: unset: name: cannot unset: readonly variable
```

2. Read-Only Constants

Read-only variables are particularly useful for constants:

```
#!/bin/bash
# Define constants
readonly PI=3.14159
readonly SERVER_URL="https://api.example.com"
# Attempt to modify constants (will result in errors)
PI=3.14  # Error
SERVER_URL="https://new.example.com"  # Error
```

3. Using unset with Read-Only Variables

Read-only variables cannot be unset:

```
#!/bin/bash
# Declare a variable and make it read-only
name="ShellScripting"
readonly name
# Attempt to unset the variable
unset name  # This will cause an error
```

Output:

```
./script.sh: line 8: unset: name: cannot unset: readonly variable
```

How to Unset Non-Read-Only Variables

To remove a variable that is not read-only, you can use the `unset` command:

```
#!/bin/bash
# Declare a variable
name="TemporaryValue"
# Print the variable
echo "Before unset: ${name}"
# Unset the variable
unset name
# Attempt to print the variable
echo "After unset: ${name}"  # Will print nothing
```

Output:

```
Before unset: TemporaryValue
After unset:
```

Best Practices

1. **Use for Constants:**
   - Always use `readonly` for variables that represent constants or configurations that should not be altered during script execution.
2. **Avoid Overuse:**
   - Do not use `readonly` for variables that need to be updated dynamically.
3. **Error Handling:**
   - Be cautious when modifying scripts with read-only variables, as they can cause unexpected errors if attempts are made to modify them.

Convert String in Shell Script

String manipulation is an essential part of Shell scripting. In this tutorial, we will explore various operations you can perform on strings, such as converting cases and finding the length of a string.

Convert First Character to Upper Case in Shell Script

You can convert the first character of a string to uppercase using the following syntax:

```
#!/bin/bash
string="my name is Gaurav"
echo "${string^}" # My name is Gaurav
```

Convert a String to Upper Case in Shell Script

To convert the entire string to uppercase:

```
#!/bin/bash
string="my name is Gaurav"
echo "${string^^}" # MY NAME IS GAURAV
```

Convert First Character to Lower Case in Shell Script

To convert the first character of a string to lowercase:

```
#!/bin/bash
string="My name is Gaurav"
echo "${string,}" # my name is Gaurav
```

Convert a String to Lower Case in Shell Script

To convert the entire string to lowercase:

```
#!/bin/bash
string="My name is Gaurav"
echo "${string,,}" # my name is gaurav
```

Get the Length of a String in Shell Script

To calculate the length of a string variable:

```
#!/bin/bash
string="My name is Gaurav"
echo "Length of the string is ${#string}"
```

Complete Example

Here is a complete example demonstrating all the above string manipulations:

```
#!/bin/bash
string="my name is Gaurav"
echo "Original string: ${string}"
echo "First character to uppercase: ${string^}"
echo "Entire string to uppercase: ${string^^}"
string="My name is Gaurav"
echo "Original string: ${string}"
echo "First character to lowercase: ${string,}"
echo "Entire string to lowercase: ${string,,}"
echo "Length of the string: ${#string}"
```

Output:

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./string-variable.sh
Original string: my name is Gaurav
First character to uppercase: My name is Gaurav
Entire string to uppercase: MY NAME IS GAURAV
Original string: My name is Gaurav
First character to lowercase: my name is Gaurav
Entire string to lowercase: my name is gaurav
Length of the string: 17
```

Substring in Shell Script

Substring operations in Shell scripting allow you to extract, modify, and manipulate strings effectively. This tutorial provides step-by-step examples to help you master substring operations.

Get Substring from a String

You can extract a substring starting from a specific position using the syntax:

```
${string:position}
```

Example:

```
#!/bin/bash
string="abcgauravabcxyz"
echo "${string:0}"  # abcgauravabcxyz
echo "${string:1}"  # bcgauravabcxyz
echo "${string:4}"  # gauravabcxyz
```

Get Last `n` Characters from a String

To extract the last `n` characters of a string:

```
#!/bin/bash
string="abcgauravabcxyz"
echo "${string: -3}" # xyz
```

Get Substring with Specific Length

You can extract a substring of a specific length starting from a given position:

```
#!/bin/bash
string="abcgauravabcxyz"
echo "${string:0:3}" # abc
echo "${string:3:3}" # gau
```

Get Shortest Match from Starting

To remove the shortest match from the beginning of the string:

```
#!/bin/bash
string="abcgauravabcxyz"
echo "${string#a*c}" # gauravabcxyz
```

Get Longest Match from Starting

To remove the longest match from the beginning of the string:

```
#!/bin/bash
string="abcgauravabcxyz"
echo "${string##a*c}" # xyz
```

Get Shortest Match from the End

To remove the shortest match from the end of the string:

```
#!/bin/bash
string="abcgauravabcxyz"
echo "${string%b*z}" # abcgaurava
```

Get Longest Match from the End

To remove the longest match from the end of the string:

```
#!/bin/bash
string="abcgauravabcxyz"
echo "${string%%b*z}" # a
```

Replace First Occurrence of a Substring

To replace the first occurrence of a substring:

```
#!/bin/bash
string="abcgauravabcxyz"
echo "${string/abc/xyz}" # xyzgauravabcxyz
```

Replace All Occurrences of a Substring

To replace all occurrences of a substring:

```
#!/bin/bash
string="abcgauravabcxyz"
echo "${string//abc/xyz}" # xyzgauravxyzxyz
```

Remove First Occurrence of a Substring

To remove the first occurrence of a substring:

```
#!/bin/bash
string="abcgauravabcxyz"
echo "${string/abc}" # gauravabcxyz
```

Remove All Occurrences of a Substring

To remove all occurrences of a substring:

```
#!/bin/bash
string="abcgauravabcxyz"
echo "${string//abc}" # gauravxyz
```

Full Script Example

Here’s a complete example demonstrating all the above operations:

```
#!/bin/bash
string="abcgauravabcxyz"
echo "${string:0}"
echo "${string:1}"
echo "${string:4}"
echo "${string:0:3}"
echo "${string:3:3}"
echo "${string: -5}"
echo "${string#a*c}"  # from starting, shortest match
echo "${string##a*c}" # from starting, longest match
echo "${string%b*z}"  # from ending, shortest match
echo "${string%%b*z}" # from ending, longest match
string="abcgauravabcxyz"
echo "${string/abc/xyz}"
echo "${string//abc/xyz}"
echo "${string/abc}"
echo "${string//abc}"
```

Output

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./substring.sh
abcgauravabcxyz
bcgauravabcxyz
gauravabcxyz
abc
gau
bcxyz
gauravabcxyz
xyz
abcgaurava
a
xyzgauravabcxyz
xyzgauravxyzxyz
gauravabcxyz
gauravxyz
```

Set Default Value to Shell Script Variable

If parameter is unset or null Set Default Value

`${parameter:-word}`

If parameter is unset or null, the expansion of word is substituted. Otherwise, the value of parameter is substituted.

If parameter is unset then Set Default Value

```
${parameter-word}
```

If parameter is unset, the expansion of word is substituted. Otherwise, the value of parameter is substituted.
let's create a script and execute it for practice purpose.

```
#!/bin/bash
read -p " please enter your name " name
name=${name:-World}
echo "Hello ${name^}"
yourname=${unsetvariable-Manish}
echo $yourname
myname=""
mytestname=${myname:-kali}
echo ${mytestname}
```

output:

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./defaultvalue.sh
 please enter your name Gaurav
Hello Gaurav
Manish
kali
```

Check A Variable is set or not using below script

```
#!/bin/bash
# name="gaurav"
: ${1:?" please set variable values. "}
echo "i am here."
```

output:

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./if-variable-not-set.sh Gaurav
i am here.

┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./if-variable-not-set.sh
./if-variable-not-set.sh: line 4: 1:  please set variable values.
```

Arithmetic Operations in Shell Script

We can do arithmetic operations in shell script in a serval way (using let command, using expr command) but we will recommend using brackets for that.

Different ways to compute Arithmetic Operations in Bash

1. Using Double Parenthesis
2. Using let command
3. using expr command

Using Double Parenthesis

Addition

```
Sum=$((20+2))
echo "Sum = ${Sum}" # output will be 22
```

Subtraction

```
sub=$((29-2))
echo "sub = ${sub}" # output will be 27
```

Multiplication

```
mul=$((20*4))
echo "Multiplication = ${mul}"  # output will be 80
```

Division

```
div=$((10/3))
echo "Division = ${div}" # output will be 3
```

Modulo

```
mod=$((10%3))
echo "Modulo = ${mod}"  # output will be one.
```

Exponentiation

```
exp=$((10**2))
echo "Exponent = ${exp}" # output will be 100.
```

let's create a shell script that will perform some arithmetic operations and some increment and decrement operations.

```
#!/bin/bash
a=5
b=6
echo "$((a+b))"
echo "$((a-b))"
echo "$((a*b))"
echo "$((a/b))" # 5/6
echo "$((a%b))"
echo "$((2**3))" # 2*2*2
((a++)) # a=a+1
echo $a
((a+=3)) # a=a+3
echo $a
```

output

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./arth-operator.sh
11
-1
30
0
5
8
6
9
```

Using let Command

let command is used to perform arithmetic operations.

```
#!/bin/bash
x=10
y=3
```

```
let "z = $(( x * y ))"  # multiplication
echo $z
let z=$((x*y))
echo $z
let "z = $(( x / y ))"  # division
echo $z
let z=$((x/y))
echo $z
```

output

```
30
30
3
3
```

expr command with backticks

The arithmetic expansion could be done using backticks and expr.

```
#!/bin/bash
a=10
b=3
# there must be spaces before/after the operator
sum=`expr $a + $b`
echo $sum
sub=`expr $a - $b`
echo $sub
mul=`expr $a \* $b`
echo $mul
div=`expr $a / $b`
echo $div
```

Output:

```
13
7
30
3
```