SHELLSCRIPTING

By: Jaswanth T

—

Notes

📄 Introduction to Shell Scripting Basics

💠 What is Shell Scripting?

Shell scripting is a powerful way to automate and streamline tasks within a Linux environment by writing and executing commands. It's invaluable for system administration, task automation, and enhancing productivity.

🔑 **Types of Shells:** Some popular shells include:

- `bash` (Bourne Again Shell) - most widely used
- `sh` (Bourne Shell)
- `ksh` (KornShell)
- `zsh` (Z Shell)

💡 **Check Your Shell Type:** Run the following command to identify your current shell:

```
echo $0
```

💠 **How to Create a File and Use It?**
Creating and editing files using the `vi` command is a core skill for working in Unix and Linux environments. Here's a step-by-step guide on creating a file, entering content, and saving it with `vi`, as well as how to run your script and view its output.

1. **Opening `vi` to Create a New File** ○ Open your terminal.

   Type the following command to create or open a file in `vi`:
   ```
   vi my_script.sh
   ```

   ○ *Note: If `my_script.sh` doesn't exist, `vi` will create a new file with that name.*

2. **Basic `vi` Editing Commands**
   When you open `vi`, you're in command mode, where you can issue commands to `vi`. Here's how to enter insert mode to start writing:
   ○ Press `i` to enter insert mode. You can now type content into the file.

3. **Writing Content in `vi`**
   Let's create a simple script that prints "Hello, World!" and displays the current date and time. With `vi` in insert mode, type the following script:

   ```
   #!/bin/bash

   # This script prints a greeting and the current date
   and time
   ```

```
echo "Hello, World!" echo "The
current date and time is:" date
```

4. **Exit Insert Mode:**
   Press `Esc` to return to command mode.

5. **Saving and Exiting in `vi`**
   To save the file and exit `vi`:
   - Type `:wq` and press Enter. `:wq` means "write (save) and quit." Alternatively:
   - To save without exiting, type `:w` and press Enter.
   - To quit without saving changes, type `:q!` and press Enter.

6. **Making the Script Executable**
   Before running your script, you need to make it executable. In the terminal, type:
   ```
   chmod +x my_script.sh
   ```

7. **Running the Script and Viewing Output**
   Run the script by typing:
   ```
   ./my_script.sh
   ```

8. **Run the Script Without Making It Executable:**
   You can directly run it with `bash` by typing:
   ```
   bash my_script.sh
   ```
   View the Output:

When you execute the script, you should see output similar to this:

```
Hello, World!

The current date and time is:

Tue Nov 6 12:34:56 UTC 2023
```

💠 Process Control Shortcuts

- `Ctrl + C`: Terminates a running process immediately. Useful for stopping commands or scripts that are running.
- `Ctrl + Z`: Pauses (stops) a process and sends it to the background. Resume with `fg` (foreground) or `bg` (background).

💠 Comments in Shell Scripting

Single-Line Comments:
Use the `#` symbol at the beginning of a line to create a single-line comment.

```
# This is a single-line comment

echo "Hello, World!" # This comment is on the same line as a command
```

Multi-Line Comments:
Bash does not have a direct syntax for multi-line comments, but you can simulate it using a here-document with `<<`. `: <<'COMMENT'`

```
This is a multi-line comment.

You can add multiple lines here.

COMMENT
```

Alternatively, use the `<<` syntax directly:

```
<<comment

This is a multi-line comment.

It won't be executed.

comment
```

📜 Writing Your First Shell Script 💡

Getting Started with Scripting

A script is a series of commands executed sequentially. Here's a simple example that prints a message to the screen:

```
#!/bin/bash # Shebang to specify the interpreter echo

"Hello World!"
```

Output:

```
Hello World!
```

🖥 How to Run Your Script:

Make it Executable:

```
chmod +x script.sh
```

Execute the Script:

```
./script.sh
```

Why Shebang (`#!/bin/bash`) Matters

This line tells the system to use Bash to run the script, ensuring compatibility across Linux environments.

📊 Working with Variables and Arrays

Variables and arrays allow data storage and manipulation in scripts, making automation more flexible.

◈ **Variables:** Define a variable and use `$` to access its value.

```
NAME="Linux"

echo "Welcome to $NAME Scripting"
```

◈ **Constants:** Make variables read-only with `readonly`.

```
readonly VERSION="1.0"
```

◈ **Arrays:** Store multiple values.

```
myArray=(1 2 3 "Hello" "World")

echo "${myArray[1]}"

Outputs: 2
```

- **Get the length of an array:**

  `echo "${#myArray[@]}" or echo "${#myArray[*]}"`

  This returns the total number of elements in the array.

- **Get specific values from an array:** To get values starting from a specific index: `echo "${myArray[*]:1}"`

  This fetches all elements starting from the second element (index 1).

- **To get a specific range of values:** `echo "${myArray[*]:1:2}"`

  This fetches 2 elements starting from index 1.

- **Update an array (Add new elements):**

  ```
  myArray+=(5 6 8)
  ```

- **Working with Associative Arrays**

  Declare and Initialize: `declare -A myArray myArray=( [name]=Paul [age]=20 )`

  Access Values: `echo "${myArray[name]}"`

  Get Array Length:
  `echo "${#myArray[@]}"`

  Update Array: `myArray+=( [city]=NewYork )`

💡 Arrays are handy for managing lists of values, such as filenames or configurations.

🧩 String and Arithmetic Operations

String Manipulation Examples: `str="Shell Scripting"`

Length:

`echo ${#str} Outputs: 15`

Replace:
`echo ${str/Scripting/Programming} Outputs: Shell Programming`

Extract Substring:
`echo ${str:6:9} Outputs: Scripting`

Get the length of a string:
`myVar="Hello World!" length=${#myVar} echo $length Output: 12`

Convert to uppercase: `upper=${myVar^^}`
`echo $upper Output: HELLO WORLD!`

Convert to lowercase: `lower=${myVar,,}`
`echo $lower Output: hello world!`

Replace a substring:
`replace=${myVar/World/Buddy} echo $replace Output: Hello Buddy!`

Extract a substring (slice):
`slice=${myVar:6:5} echo $slice Output: World` ◈ **USER INTERACTIONS**

Taking Input from the User in Shell Scripting

- **Basic Input**: `read var_name echo "You entered: $var_name"`

Example Output:

(User types "John")

You entered: John

- **Input with Prompt**:

```
read -p "Your name: " NAME
echo "Hello, $NAME!"
```

Example Output:

(Your name: User types "John")

Your name: John

Hello, John!

Key Difference:

- **Basic Input**: Takes input without a prompt.
- **Input with Prompt**: Displays a prompt to guide the user.

◈ Arithmetic Operations

- **Using the `let` command**:

  Increment:

  ```
      let a++
  ```

  This increments the value of `a` by 1.

  Assignment with multiplication:

  ```
      let a=5*10
  echo $a Output: 50
  ```

- **Using `(( ))` for arithmetic operations**:

  Increment:

```
((a++))
```

Assignment with multiplication:

```
((a=5*10))
echo $a Output: 50
```

Key Difference:

- `let` is more traditional, while `(( ))` is more modern and allows for more complex arithmetic expressions.

Use `$((expression))` for complex calculations:

```
echo $((5 * (3 + 2))) Output: 25
```

☑ Conditional Statements in Shell Scripting if

Statement:

```
if [ $a -gt $b ]; then echo "a

is greater than b" fi
```

if-else Statement:

```
if [ $a -gt $b ]; then

    echo "a is greater than b"

else echo "a is not greater than

b" fi
```

elif (else if) Statement:

```
if [ $a -gt $b ]; then echo "a
    is greater than b"
elif [ $a -eq $b ]; then
    echo "a is equal to b"
else
    echo "a is less than b"
fi
```

Case Statement:

```
case $a in
    a) echo "a is 1" ;;
    b) echo "a is 2" ;;
    *) echo "a is neither 1 nor 2" ;;
esac
```

Key Notes:

- Always put spaces around operators in conditions.
- `elif` and `else` are optional but useful for handling multiple conditions.

Comparison Operators

Equal to: `-eq` or `==`: Checks if two values are equal.

```
[ $a -eq $b ]
```

Greater Than or Equal to: `-ge`: Checks if the left operand is greater than or equal to the right.

```
[ $a -ge $b ]
```

Less Than or Equal to: `-le`: Checks if the left operand is less than or equal to the right.

```
[ $a -le $b ]
```

Not Equal to: `-ne` or `!=`: Checks if two values are not equal.

```
[ $a -ne $b ]
```

Greater Than: `-gt`: Checks if the left operand is greater than the right.

```
[ $a -gt $b ]
```

Less Than: `-lt`: Checks if the left operand is less than the right.

```
[ $a -lt $b ]
```

◈ Logical Operators

Using `&&` (AND) Operator:

```
a=10
b=5
if [ $a -gt 5 ] && [ $b -lt 10 ]; then
    echo "Both conditions are true"
else echo "One or both conditions are
    false"
fi
```

Using `||` (OR) Operator:

```
a=10
b=15
if [ $a -gt 5 ] || [ $b -lt 10 ]; then
    echo "At least one condition is true"
else echo "Neither condition is
    true"
fi
```

Combining `&&` and `||` Operators:

```
a=10
b=5
c=15
if [ $a -gt 5 ] && [ $b -lt 10 ] || [ $c -eq 15 ]; then
    echo "Condition met"
else echo "Condition not
    met"
fi
```

Explanation:

- The `&&` ensures both conditions must be true.
- The `||` checks the second condition if the first fails.

◈Ternary Operator (One-liner If-Else)

A simple way to write an `if-else` statement in one line:

```
a=10
[ $a -gt 5 ] && echo "Greater" || echo "Not Greater"
```

Explanation:

● If `$a -gt 5` is true, it prints "Greater". ● Otherwise, it prints "Not Greater".

🔄 For Loop

The `for` loop iterates over a list or a range of values and performs actions for each item.

Syntax:

```
for item in list; do
  # Commands to execute for each item
done
```

Example:

```
for i in 1 2 3; do
  echo "Number: $i"
done
```

Output:

```
Number: 1
Number: 2
Number: 3
```

Range Example:

```
for i in {1..3}; do
  echo "Count: $i"
done
```

Output:

```
Count: 1
Count: 2
Count: 3
```

🔄 While Loop

The `while` loop runs as long as the specified condition is true.

Syntax:

```
while [ condition ]; do
  # Commands to execute
done
```

Example:

```
count=1
while [ $count -le 3 ]; do
  echo "Count is: $count"
  ((count++)) # Increment count done
```

Output:

```
Count is: 1
Count is: 2
Count is: 3
```

🔄 Until Loop

The `until` loop continues to execute until the condition becomes true.

Syntax:

```
until [ condition ]; do
  # Commands to execute
done
```

Example:

```
count=1
until [ $count -gt 3 ]; do
  echo "Count is: $count"
  ((count++))
done
```

Output:

```
Count is: 1
Count is: 2
Count is: 3
```

🔄 Infinite Loop

An infinite loop continues running indefinitely until it is manually stopped (e.g., using Ctrl+C).

For Loop Infinite Example:

```
for (( ; ; )); do

  echo "This is an infinite loop"

done
```

While Infinite Example:

```
while :; do

  echo "Infinite loop with while"

done
```

Until Infinite Example:

```
until false; do

  echo "Infinite loop with until"

done
```

Output (repeats indefinitely):

```
This is an infinite loop

This is an infinite loop

...
```

🗒 Select Loop

The `select` loop creates a simple menu system, which allows users to select an option from a list. It's useful when you need a user-driven selection process.

Syntax:

```
select option in list; do
  # Commands based on user choice
done
```

Example:

```
PS3="Choose a fruit: " select fruit in Apple
Banana Orange Exit; do
  case $fruit in
    Apple) echo "You chose Apple";;
    Banana) echo "You chose Banana";;
    Orange) echo "You chose Orange";;
    Exit) break;;
    *) echo "Invalid option";;
  esac
done
```

Example Output:

```
1) Apple
2) Banana
3) Orange
4) Exit
```

```
Choose a fruit: 2
You chose Banana
```

Explanation:

- `PS3` sets the prompt message.
- The `select` loop displays options, and each selection runs the corresponding case statement.
- The `break` statement exits the loop when the user selects "Exit."

Summary of Loop Types

- **For Loop**: Iterates over a list or range.
- **While Loop**: Continues as long as a condition is true.
- **Until Loop**: Continues until a condition becomes true.
- **Infinite Loop**: Runs indefinitely until interrupted.
- **Select Loop**: Displays a menu for user selection.

Tip:

Loops are powerful tools for automating repetitive tasks. You can use them for various purposes like iterating over files, arrays, or ranges. For example, you can rename all files in a directory using a loop.

Functions

1. **Defining Functions:** You can define a function using either of these two

   syntaxes:

   ```
   function function_name { ... }
   ```
   or
   ```
   function_name() { ... }
   ```

2. **Basic Function:** Functions are used to encapsulate reusable blocks of code.

```
greet() {
  echo "Hello, welcome to the shell script!"
}
greet # Calling the function
```

3. **Functions with Parameters:** Functions can accept arguments, which are accessed via $1, $2, etc.

```
greet_user() {
echo "Hello, $1!"
}
greet_user "Adhyansh"
```

4. **Return Values:** Functions return values via echo, and the output can be captured.

```
add_numbers() {
 result=$(( $1 + $2 ))
 echo $result
} sum=$(add_numbers 3
5) echo "The sum is:
$sum"
```

5. **Conditional Logic and Loops:** Functions can include conditions and loops.

```
check_even() {
```

```
if (( $1 % 2 == 0 ));
then echo "$1 is even"
else echo "$1 is odd" fi
}
check_even 7 output: "7 is odd"
```

6. **Recursion:** Functions can call themselves recursively.

```
factorial() {
if [ $1 -le 1 ]; then
     echo 1
else prev=$(factorial $(( $1 - 1
     )))
     echo $(( $1 * prev ))
fi
}
re
su
lt
=$
(f
ac
to
ri
al
5)
ec
ho
"F
ac
to
ri
al
of
```

```
5
is
:
$r
es
ul
t"
```

7. **Default Values:**

   Set default values for arguments using `${1:-default_value}`.

   ```
   greet() {
    local name=${1:-Guest}
    echo "Hello, $name!"
   }
   greet "Adhyansh" Output: "Hello, Adhyansh!" greet
   Output: "Hello, Guest!"
   ```

8. **Passing Arguments by Reference:** Bash doesn't directly support

   passing by reference but can simulate it using `eval`.

   ```
   modify_value() {
       eval $1=\$2
   }
   modify_value var 100
   echo "The value of var is now: $var" Output: "The value
   of var is now: 100"
   ```

9. **Argument Passing:**

   You can pass arguments to functions, and they can be accessed inside the function.

- **Positional Arguments**: $1, $2, $3, etc. (Access individual arguments).
- **All Arguments**: $@ (all arguments as separate words), $* (all arguments as a single string).
- **Argument Count**: $# (number of arguments passed).

1. **Positional Arguments ($1, $2, $3, etc.)**

- These represent individual arguments passed to a function or script. • $1 is the first argument, $2 is the second, and so on. **Example**:

```
greet() {
  echo "Hello, $1! You are $2 years old."
} greet "Adhyansh"
25
```

```
Output: Hello, Adhyansh! You are 25 years old.
```

2. **All Arguments ($@)**

- This represents all the arguments passed to a function or script.
- Each argument is treated as a separate word, which is especially useful when looping over the arguments. **Example**:

```
print_all() { for arg
  in "$@"; do
    echo "$arg"
  done } print_all "Apple" "Banana"
"Cherry" Output: Apple
 Banana
 Cherry
```

3. **All Arguments as a Single String ($*)**

- Similar to $@, but it treats all arguments as a single string, meaning spaces between arguments may be lost. **Example**:

```
print_all_as_string() {
  echo "$*" } print_all_as_string "Apple"
"Banana" "Cherry" Output: Apple Banana Cherry
```

4. Number of Arguments ($#)

- This gives the count of the arguments passed to a function or script.

Example:

```
count_args() {
  echo "Number of arguments: $#"
} count_args "Apple" "Banana"
"Cherry" Output: Number of
arguments: 3
```

Summary

- Functions in shell scripts help organize code, making it modular and reusable.
- Arguments can be passed to functions, accessed via $1, $2, etc., and default values can be set.
- You can use loops and conditionals inside functions to add logic.
- Functions return values using `echo`, which can be captured in variables.

🔄 shift Operator in Bash

The `shift` command in Bash allows you to move (shift) the positional parameters (arguments) passed to a script or function. Here's a brief explanation of how it works:

Syntax:

- `shift`: Shifts positional parameters left by one position.
- `shift n`: Shifts positional parameters left by `n` positions.

Example 1: Shifting Arguments (One Position)
```
shift_example() {
  echo "Original arguments: $1, $2, $3"
  shift
  echo "After shift: $1, $2, $3"
} shift_example "one" "two"
"three"
```

Output:
```
Original arguments: one, two, three
After shift: two, three
```
After the `shift`, `$1` becomes "two", and `$2` becomes "three".

Example 2: Shifting Arguments Multiple Times
```
shift_multiple() {
  echo "Original arguments: $1, $2, $3, $4"
  shift 2
  echo "After shifting 2 times: $1, $2"
}
shift_multiple "apple" "banana" "cherry" "date"
```
Output:
```
Original arguments: apple, banana, cherry, date
After shifting 2 times: cherry, date
```
After shifting 2 positions, "apple" and "banana" are discarded, and the remaining arguments are shifted left.

Example 3: Using `shift` with a Loop
```
process_args()
{
  while [ "$#" -gt 0 ]; do
    echo "Processing argument: $1"
    shift
  done } process_args "arg1" "arg2"
"arg3" "arg4"
```

Output:

```
Processing argument: arg1
Processing argument: arg2
Processing argument: arg3
Processing argument: arg4
```

The loop processes each argument one by one by shifting the arguments left until there are no more arguments (`$#` becomes 0).

Example 4: Combining `shift` with `$#` to Count Remaining Arguments

```
process_first_two() {
  echo "First argument: $1"
  echo "Second argument: $2"
  shift 2
  echo "Remaining arguments: $@"
}
process_first_two "apple" "banana" "cherry" "date"
```

Output:

```
First argument: apple
Second argument: banana
Remaining arguments: cherry date
```

After shifting by 2, the first two arguments are removed, and the remaining arguments are accessible using `$@`.

Key Points:

- `shift` removes the first argument (`$1`) and shifts the remaining arguments left.
- You can shift by multiple positions using `shift n`.
- The `$#` variable always reflects the remaining number of arguments.
- The `shift` command is useful in loops for processing a variable number of arguments.

---

break Statement

- **Purpose**: Exits a loop prematurely.
- **Syntax**: `break` or `break n` (exits the loop and can exit multiple nested loops with `n`). **Example**:

```
for i in {1..5}; do
  if [ $i -eq 3 ]; then break; fi
  echo $i
done
```

- **Output**: Loops until `i=3`, then exits.

2. continue Statement

- **Purpose**: Skips the current loop iteration and moves to the next one.
- **Syntax**: `continue` or `continue n` (skips current iteration in nested loops). **Example**:

```
for i in {1..5}; do
  if [ $i -eq 3 ]; then continue; fi
  echo $i
done
```

- **Output**: Skips iteration when `i=3`.

3. break and continue in Nested Loops

- **break**: `break n` can exit multiple levels of nested loops.
- **continue**: `continue n` skips the current iteration in multiple nested

loops. **Example**:

```
for i in {1..3}; do
  for j in {1..3}; do if [ $i -eq 2 ] && [ $j -eq 2
    ]; then break 2; fi
    echo "i=$i, j=$j"
  done
done
```

- **Output**: Exits both loops when `i=2` and `j=2`.

---

4. sleep Command

- **Purpose**: Pauses the script for a specified time.
- **Syntax**: `sleep <duration>` **Example**:

```
sleep 5 # Pauses for 5 seconds
```

5. exit Command

- **Purpose**: Exits a script with a status code (0 for success, non-zero for error).
- **Syntax**: `exit <exit_code>` **Example**:

```
exit 1 # Exits with an error status
```

6. $? (Exit Status of Last Command)

- **Purpose**: Stores the exit status of the last executed command. **Example**:

```
mkdir myfolder echo $? # Returns 0 if
successful, 1 if failed.
```

Summary

- **break**: Exit loops.
- **continue**: Skips an iteration in loops.
- **sleep**: Pauses script execution.
- **exit**: Terminates a script with a status code.
- **$?**: Checks the exit status of the last command.

`seq` command

The `seq` command in Linux is used to generate a sequence of numbers. It's commonly used in shell scripting or on the command line to generate numbers for loops, lists, or ranges.

Basic Syntax:

```
seq [options] <start> <end>
```

● **start**: The starting number of the sequence. ●
end: The ending number of the sequence.

Examples of Using the `seq` Command: Basic Sequence: Generate

a sequence of numbers from 1 to 5: `seq 1 5`

Output:

```
1
2
3
```

```
4
5
```

Custom Step Size: You can specify a step size between numbers. For example, to generate numbers from 1 to 10 with a step size of 2: `seq 1 2 10`

Output:

```
1
3
5
7
9
```

Generating a Sequence with Decimal Values: You can also use floating-point numbers with `seq`. For example, generate numbers from 0 to 1 with a step size of 0.2:

```
seq 0 0.2 1
```

Output:

```
0.0
0.2
0.4
0.6
0.8
1.0
```

Reverse Sequence: You can reverse the order of the sequence by making the start number greater than the end number: `seq 5 -1 1`

Output:

```
5
4
3
2
1
```

Using `seq` with `for` Loop: You can use `seq` in a `for` loop. For example, printing numbers from 1 to 5:

```
for i in $(seq 1 5); do
    echo "Number: $i"
done
```

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

Common Options with `seq`:

-f: Format the output using a format string (like `printf`). `seq -f "Number: %.2f" 1 5`

Output:

```
Number: 1.00
Number: 2.00
Number: 3.00
Number: 4.00
Number: 5.00
```

-s: Change the separator between numbers. By default, it's a newline, but you can change it:

```
seq -s "," 1 5
```

Output:

```
1,2,3,4,5
```

-w: Pad numbers with leading zeros to make them all the same width: `seq -w 1 5`

Output:

```
01
02
03
04
05
```

Summary:

The `seq` command is a powerful and simple tool to generate sequences of numbers, and it can be customized in many ways using options for formatting, step size, and separators.

Brace Expansion is a feature in the shell (especially Bash) that allows you to generate multiple text strings from a single expression. It is typically used to create lists of strings or numbers, or to generate filenames with multiple variations. Brace expansion is a simple and efficient way to generate multiple values without having to manually list them.

Basic Syntax:

```
{item1,item2,item3}
```

This will generate a list containing the individual items within the braces.

Examples of Using Brace Expansion:

Creating a List of Words: You can generate a list of words by expanding a comma-separated list inside braces:

```
echo {apple,banana,cherry}
```

Output:

```
apple banana cherry
```

Range of Numbers: You can create a sequence of numbers by using a range inside the braces:

```
echo {1..5}
```

Output:

```
1 2 3 4 5
```

Range of Letters: You can also create a range of letters:

```
echo {a..e}
```

Output:

```
a b c d e
```

Range with Step Size: You can specify a step size by including it after the range:

```
echo {1..10..2}
```

Output:

```
1 3 5 7 9
```

Combining Text with Variables: You can use brace expansion to combine static text with variables:

```
echo file{1..3}.txt
```

Output:

```
file1.txt file2.txt file3.txt
```

Nested Brace Expansion: You can also nest brace expansions to create more complex patterns:

```
echo {A,B}{1,2}
```

Output:

```
A1 A2 B1 B2
```

Multiple Elements in a Single Expansion: You can combine multiple sets of values within a single brace expansion:

```
echo {A,B,C}{1,2,3}
```

Output:

```
A1 A2 A3 B1 B2 B3 C1 C2 C3
```

Files and Directories: Brace expansion can also be used to generate multiple file or directory names:

```
mkdir dir{1,2,3}
```

This will create the directories `dir1`, `dir2`, and `dir3`.

Important Points:

- **Brace expansion happens before any other shell operation** like variable expansion or command substitution.
- It is **not the same as parameter expansion or globbing**. It is handled directly by the shell before the command is executed.
- **No spaces between commas in the braces**. It should be `{item1,item2}` instead of `{item1, item2}`.

Summary:

Brace expansion is a powerful feature in the shell that allows you to generate sequences of strings, numbers, or characters without writing them out explicitly. It can save time and reduce errors when generating repetitive patterns, filenames, or arguments.

`getopts` command

The `getopts` command is used in shell scripting (primarily in Bash) to parse command-line options (flags) and arguments passed to a script. It provides a way to handle options and arguments in a structured and consistent manner, making your script more flexible and user-friendly.

Basic Syntax:

```
getopts "option_string" variable
```

- `option_string`: A string of valid option letters. If an option requires an argument, you append a colon (`:`) after the letter (e.g., `a:` means option `a` expects an argument).
- `variable`: The variable where the parsed option will be stored.

How `getopts` Works:

- `getopts` processes command-line arguments one by one.
- It sets the value of the `variable` to the current option letter.
- If the option requires an argument, `getopts` will assign that argument to a special variable, usually `$OPTARG`.

Example 1: Simple Options Parsing

This example shows how to handle single-letter options, such as `-a` or `-b`, in a script.

Script Example (`myscript.sh`):

```
#!/bin/bash
while getopts "ab" option; do
  case $option in
    a) echo "Option A selected" ;;
    b) echo "Option B selected" ;;
    \?) echo "Invalid option"; exit 1 ;;
  esac
```

```
done
```
Running the script:

```
$ ./myscript.sh -a
Option A selected

$ ./myscript.sh -b
Option B selected
```

Example 2: Options with Arguments

You can also specify options that require arguments (e.g., `-f filename`).

Script Example (`myscript.sh`):

```
#!/bin/bash
while getopts "f:n:" option; do
  case $option in
    f) echo "Option F selected with argument: $OPTARG" ;;
    n) echo "Option N selected with argument: $OPTARG" ;;
    \?) echo "Invalid option"; exit 1 ;;
  esac
done
```

Running the script:

```
$ ./myscript.sh -f file.txt
Option F selected with argument: file.txt

$ ./myscript.sh -n 123
Option N selected with argument: 123
```

Example 3: Handling Long Options with Arguments

For options that have both short and long forms, you can use `getopts` to handle short options (e.g., `-f`) but long options require additional parsing logic. Here's an example for a simple long-option-like approach.

Script Example (`myscript.sh`):

```
#!/bin/bash
while getopts "f:n:" option; do
  case $option in
    f) echo "File option: $OPTARG" ;;
    n) echo "Name option: $OPTARG" ;;
    \?) echo "Invalid option"; exit 1 ;;
  esac
done
```

Running the script:

```
$ ./myscript.sh -f myfile.txt -n John
File option: myfile.txt
Name option: John
```

Explanation of Common `getopts` Usage:

- **`$OPTARG`**: This variable contains the value of the argument passed to an option (if any).
- **`$option`**: The current option being processed.
- **`\?`**: Used in the `case` statement to catch invalid options.

Example 4: Parsing Multiple Arguments and Options

In this example, we handle multiple options that require arguments:

Script Example (`myscript.sh`):

```
#!/bin/bash
```

```
while getopts "a:b:c:" option; do
  case $option in
       a) echo "Option A selected with value: $OPTARG" ;;
       b) echo "Option B selected with value: $OPTARG" ;;
       c) echo "Option C selected with value: $OPTARG" ;;
    \?) echo "Invalid option"; exit 1 ;;
  esac
done
```

Running the script:

```
$ ./myscript.sh -a 10 -b 20 -c 30
Option A selected with value: 10
Option B selected with value: 20
Option C selected with value: 30
```

Exiting When No Arguments Are Provided:

If no arguments are given or the `getopts` finds a missing or incorrect argument, it will return `?`, which can be caught to display an error or usage message.

Example: `#!/bin/bash`

```
if [ $# -eq 0 ]; then

  echo "No options provided."

  exit 1

fi

while getopts "a:b:" option; do

  case $option in

    a) echo "Option A with argument: $OPTARG" ;;
    b) echo "Option B with argument: $OPTARG" ;;
    \?) echo "Invalid option"; exit 1 ;;
  esac
done
```

Key Points:

- **`getopts` is used to process options one by one**.
- **`$OPTARG` contains the value for options that take arguments**.
- **`getopts` does not handle long options (e.g., `--option`)** directly; you'll need to process them manually if needed.
- **Return codes**:
  - `0`: The option was successfully processed. ○ `1`: Invalid option or missing argument.

Advantages:

- Provides a simple and standard way to handle command-line options. ● Makes scripts more user-friendly and interactive.

---

1. **basename:**

- Strips the directory path and returns only the filename. You can also remove a file extension if specified.
- **Example**: `basename /home/user/file.txt` → `file.txt`, and `basename /home/user/file.txt .txt` → `file`.

2. **dirname:**

- Strips the filename and returns the directory path.
- **Example**: `dirname /home/user/file.txt` → `/home/user`.

3. **realpath:**

- Resolves and returns the absolute path of a file or directory, resolving symbolic links.
- **Example**: `realpath file.txt` → `/home/user/Documents/file.txt`.

4. **File/Directory Existence Checks:**

- `-d <folder>`: Checks if a directory exists.
- `! -d <folder>`: Checks if a directory does not exist.
- `-f <file>`: Checks if a file exists.
- `! -f <file>`: Checks if a file does not exist. **Examples**:

```
if [ -d /home/user/Documents ]; then echo "Directory
exists."; fi

if [ ! -f /home/user/nonexistentfile.txt ]; then echo "File
does not exist."; fi
```

5. **RANDOM & UID:**

- `RANDOM`: Generates a random number between 0 and 32767.
- `UID`: Stores the user ID of the currently logged-in user. **Examples**:

```
echo $RANDOM  # Generates a random number echo $((RANDOM

% 101)) # Random number between 0 and 100 echo $UID    #

User ID of the logged-in user
```

These commands are essential tools for file path manipulation, file existence checking, and working with system information in Bash scripts. Let me know if you'd like further clarification or examples!

To continue running even after the terminal is closed, you can use **nohup** (short for "no hang up"). **This command is useful for running long-running processes or scripts in the background,** even if the terminal session is closed or disconnected. ❖ **How to Use nohup:**

To run a script in the background and keep it running after closing the terminal, use the **nohup** command followed by the script or command and an `&` to run it in the background. **nohup** `./myscript.sh &`

Example:

```
nohup bash myscript.sh &
```

This will run `myscript.sh` in the background, and the terminal can be closed without interrupting the execution of the script. The standard output will be redirected to a file called `nohup.out` unless you specify a different file.

❖ **Redirecting Output:**

If you don't want to see the output in the `nohup.out` file, you can redirect it to `/dev/null`.

```
nohup ./myscript.sh > /dev/null 2>&1 &
```

❖ **Checking Running Processes:**

You can check if the script is running by using the `ps` command or `jobs` command.

```
ps aux | grep myscript.sh
```

Or:

```
jobs
```

❖ **Stopping Background Jobs:**

To stop a background job, you can use the `kill` command with the process ID (PID) of the job.

```
kill <PID>
```

Alternatively, you can bring the background job to the foreground using `fg` and then stop it with `Ctrl + C`:

```
fg %1 # Bring job 1 to the foreground
```

Summary:

- `nohup` is used to run a command or script that will continue executing even after the terminal is closed.
- The `&` operator runs the command in the background.
- You can redirect output to a file or `/dev/null` to suppress output.
- Use `jobs` and `ps` to check the status of background jobs.

Project: Simple Calculator

This calculator will support basic arithmetic operations like addition, subtraction, multiplication, and division. The user can input the operation and two numbers, and the script will return the result.

Steps:

1. **Create a new file for the script:** Open a terminal and create a new file named `calculator.sh` using a text editor.

   ```
   vi calculator.sh
   ```
2. **Add the script code:** Below is the code for the calculator:

```
#!/bin/bash
# Function to add two numbers
add() { echo "Result: $(($1 +
$2))"
}


# Function to subtract two numbers
subtract() { echo "Result: $(($1 -
$2))"
}


# Function to multiply two numbers
multiply() { echo "Result: $(($1 *
$2))"
}


# Function to divide two numbers
divide() { if [ $2 -eq 0 ]; then echo "Error:
    Division by zero is not allowed!"
    else echo "Result: $(($1 /
        $2))"
    fi
}
# Menu to display
options echo "Simple
Calculator" echo "Choose
operation:" echo "1.
Add" echo "2. Subtract"
```

```
echo "3. Multiply" echo
"4. Divide"

read -p "Enter choice (1/2/3/4): "
choice read -p "Enter first number: "
num1 read -p "Enter second number: "
num2



# Perform the operation based on user's choice

case $choice in

    1)   add $num1 $num2
        ;;
    2)   subtract $num1
    $num2
        ;;
    3)   multiply $num1
    $num2
        ;;
    4)   divide $num1
    $num2
        ;;
    *) echo "Invalid choice"
        ;;
esac
```

3. **Make the script executable:** After saving the script, make it executable by running:

   ```
   chmod +x calculator.sh
   ```

4. **Run the script:** Now you can run the calculator script using:

```
./calculator.sh
```

How the Script Works:

- The user is prompted to select an operation (addition, subtraction, multiplication, or division).
- The script then asks for two numbers and performs the chosen operation.
- If division is selected and the second number is 0, the script will display an error message instead of attempting to divide by zero.

Example Output:

```
$ ./calculator.sh

Simple Calculator

Choose operation:

1. Add

2. Subtract

3. Multiply

4. Divide

Enter choice (1/2/3/4): 1

Enter first number: 10
Enter second number: 5

Result: 15
```

This small project helps you understand basic conditional statements, user input handling, and mathematical operations in Bash. You can expand this project by adding more advanced operations (like square roots, exponents, etc.) or even a loop to make the calculator run repeatedly.

****