Introduction to Bash Programming

Akolzina Diana

January 21, 2024

1 Introduction

Bash (Bourne Again SHell) is a powerful command-line shell interface widely used on Linux and UNIX systems. It is not only a command interpreter but also a programming language that allows for script creation to automate tasks.

2 Basic Commands

2.1 Navigation

Bash provides several commands for navigating the filesystem:

- `cd <directory>`: Changes the current directory. Use `cd ..` to go up one directory, or `cd` to return to the home directory.
- `ls [options] [directory]`: Lists files and directories. Common options include `-l` for long format, `-a` to show hidden files, and `-h` for human-readable sizes.
- `pwd`: Displays the current directory path.

2.2 File Management

Bash provides commands for managing files and directories:

- `touch <file>`: Creates a new empty file or updates the timestamp of an existing file.
- `cp <source> <destination>`: Copies files or directories. Use `-r` for recursive copying of directories.
- `mv <source> <destination>`: Moves or renames files or directories. It's used for both moving files between directories and renaming them.
- `rm <file/directory>`: Removes files or directories. Use `-r` to remove directories and their contents, and `-f` to force removal without confirmation.

- `mkdir <directory>`: Creates a new directory.
- `rmdir <directory>`: Removes empty directories.

2.3 File Viewing and Manipulation

- `cat <file>`: Displays the content of a file, or concatenates multiple files.
- `less <file>`: Allows for scrolling through content of a file.
- `head <file>`: Displays the first few lines of a file (default 10 lines).
- `tail <file>`: Displays the last few lines of a file (default 10 lines). Useful for logs.
- `grep <pattern> <file>`: Searches for a pattern in a file and prints matching lines.
- `sort <file>`: Sorts the lines in a text file.
- `uniq <file>`: Removes duplicate lines from a sorted file.
- `wc <file>`: Counts lines, words, and characters in a file.

3 Bash Scripting Basics

3.1 Variables

Bash supports the use of variables, which are used to store and manipulate data. Variables in Bash can hold data of different types, including strings, integers, and arrays.

3.1.1 Defining Variables

Variables are defined without any data type declaration, and their type is determined by the context in which they are used.

```
#!/bin/bash
name="John Doe"  % String
age=25           % Integer
```

3.1.2 Variable Expansion

Variable expansion is used to retrieve the value stored in a variable.

```
echo "Name: $name, Age: $age"
```

3.1.3 Read-Only Variables

Variables can be made read-only using the `readonly` command, preventing their modification.

```
readonly name
```

3.1.4 Environment Variables

Environment variables provide a way to influence the behavior of software on the system. They are often used to store system-wide values, such as path settings.

```
echo $PATH
```

3.1.5 Special Variables

Bash provides several special variables, like $0 for the script name, $# for the number of arguments, and $? for the last command's exit status.

3.1.6 Positional Parameters

Positional parameters are variables automatically assigned to the input arguments of a script. $1, $2, etc., correspond to the first, second, and subsequent arguments.

```
#!/bin/bash
echo "First argument: $1"
```

3.1.7 Exporting Variables

Variables can be exported to sub-processes using the `export` command. This is often used in scripts to pass variables to child processes.

3.1.8 Arithmetic in Variables

For arithmetic operations, Bash offers several methods, including the use of double parentheses.

```
#!/bin/bash
a=5
b=3
sum=$((a + b))
echo $sum
```

3.2 Math Operations

Bash supports basic math operations, which can be performed using arithmetic expansion, and more complex operations using external tools like ‘bc‘ for floating-point arithmetic.

3.2.1 Arithmetic Expansion

Arithmetic expansion allows for basic integer arithmetic using operators like addition (+), subtraction (-), multiplication (*), division (/), and modulus (

```
#!/bin/bash
a=5
b=3

# Addition
sum=$((a + b))  % Result is 8

# Subtraction
difference=$((a - b))  % Result is 2

# Multiplication
product=$((a * b))  % Result is 15

# Division
quotient=$((a / b))  % Result is 1

# Modulus
remainder=$((a % b))  % Result is 2

echo "Sum: $sum, Difference: $difference, Product: $product, Quotient: $quotie
```

3.2.2 Floating-Point Arithmetic

Bash's built-in arithmetic does not support floating-point numbers. For floating-point arithmetic, the 'bc' tool is commonly used.

```
#!/bin/bash
c=5.75
d=2.5

# Floating-point addition
fp_sum=$(echo "$c + $d" | bc)

# Floating-point division
fp_div=$(echo "scale=2; $c / $d" | bc)

echo "Floating-point Sum: $fp_sum, Division: $fp_div"
```

In this example, 'scale=2' sets the number of decimal places in the result of division.

3.2.3 Increment and Decrement

Bash also supports incrementing and decrementing variables.

```
#!/bin/bash
e=10

# Post-increment
echo $((e++))  % Outputs 10, then increments e

# Pre-increment
echo $((++e))  % Increments e first, then outputs 12

# Post-decrement and Pre-decrement can be done similarly using --
```

3.3 Arrays

Bash supports one-dimensional indexed and associative arrays.

3.3.1 Indexed Arrays

Indexed arrays use numeric indexes. Here is an example:

```
#!/bin/bash
indexed_array=("apple" "banana" "cherry")
echo ${indexed_array[1]}  % Outputs 'banana'
```

You can iterate over an indexed array using a loop:

```
for i in "${indexed_array[@]}"; do
    echo $i
done
```

3.3.2 Associative Arrays

Associative arrays use named keys instead of numeric indexes. They are declared using ‘declare -A‘.

```
#!/bin/bash
declare -A assoc_array
assoc_array[fruit]="apple"
assoc_array[vegetable]="carrot"
assoc_array[drink]="water"
echo ${assoc_array[vegetable]}  % Outputs 'carrot'
```

To iterate over keys and values in an associative array:

```
for key in "${!assoc_array[@]}"; do
    echo "$key: ${assoc_array[$key]}"
done
```

Arrays can also be modified:

```
# Adding an element to an indexed array
indexed_array+=("date")

# Modifying an element in an associative array
assoc_array[drink]="juice"
```

Array elements can be removed using the ‘unset‘ command:

```
# Remove an element from an indexed array
unset indexed_array[1]  % Removes 'banana'

# Remove an element from an associative array
unset assoc_array[drink]
```

4 Control Structures

Control structures in Bash scripting allow for decision making, looping, and branching, enabling scripts to exhibit complex behavior based on conditions and iterations.

4.1 If Statements

The ‘if‘ statement is used for conditional execution of commands in Bash. It tests a condition and executes commands based on whether the condition is true or false.

4.1.1 Basic If Statement

A basic ‘if‘ statement syntax is as follows:

```
#!/bin/bash
if [ condition ]; then
    # commands to execute if condition is true
fi
```

4.1.2 If-Else Statement

The ‘if-else‘ statement provides an alternative set of commands that are executed when the condition is false.

```
if [ condition ]; then
    # commands if condition is true
else
    # commands if condition is false
fi
```

4.1.3 Else-If Ladder

For multiple conditions, an ‘else-if‘ ladder can be used.

```
if [ condition1 ]; then
    # commands if condition1 is true
elif [ condition2 ]; then
    # commands if condition2 is true
else
    # commands if none of the above conditions are true
fi
```

4.1.4 Nested If Statements

‘if‘ statements can be nested within other ‘if‘ statements.

```
if [ condition1 ]; then
    if [ condition2 ]; then
        # commands if both condition1 and condition2 are true
    fi
fi
```

4.1.5 Using Test Conditions

Test conditions can include checks for file attributes, string comparisons, and arithmetic comparisons. Examples:

```
if [ -f "$filename" ]; then  % Check if a file exists
if [ "$str1" = "$str2" ]; then  % String comparison
if [ $val -lt 10 ]; then  % Arithmetic comparison
```

4.2 Loops

Loops are used in Bash scripting to repeatedly execute a block of code. There are several types of loops, each with its specific use case.

4.2.1 For Loop

The ‘for‘ loop is used to iterate over a series of values or an array.

Basic syntax:

```
#!/bin/bash
for i in {1..5}; do
    echo "Number $i"
done
```

Iterating over an array:

```
#!/bin/bash
arr=("apple" "banana" "cherry")
for fruit in "${arr[@]}"; do
    echo "Fruit: $fruit"
done
```

4.2.2 While Loop

The 'while' loop continues to execute as long as the given condition is true.

Basic syntax:

```
#!/bin/bash
counter=1
while [ $counter -le 5 ]; do
    echo "Counter: $counter"
    ((counter++))
done
```

4.2.3 Until Loop

The 'until' loop is similar to the 'while' loop, but it continues until the condition becomes true.

Basic syntax:

```
#!/bin/bash
counter=1
until [ $counter -gt 5 ]; do
    echo "Counter: $counter"
    ((counter++))
done
```

4.2.4 Loop Control Commands

- `break`: Exits the loop immediately.
- `continue`: Skips the rest of the loop iteration and proceeds with the next iteration.

4.2.5 C-style For Loop

Bash also supports a C-style 'for' loop for more complex iterations.

```
#!/bin/bash
for ((i = 0; i < 5; i++)); do
    echo "Number $i"
done
```

4.3 Functions

Functions in Bash are used to encapsulate a group of commands. They allow for code reuse and better organization of complex scripts.

4.3.1 Defining Functions

Functions can be defined in two ways in Bash:
Using the ‘function‘ keyword:

```
function greet {
    echo "Hello, $1!"
}
```

Or without the ‘function‘ keyword:

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

4.3.2 Passing Arguments

Arguments can be passed to functions similarly to how they are passed to scripts. Inside the function, arguments are accessed using $1, $2, etc.

```
#!/bin/bash
greet() {
    echo "Hello, $1!"
}
greet "World"
```

4.3.3 Return Values

Functions return the exit status of the last command executed. To return a specific value, use the ‘return‘ statement.

```
add() {
    local sum=$(($1 + $2))
    return $sum
}
add 5 3
echo "Sum: $?"
```

4.3.4 Local Variables

Variables within functions can be made local to the function using the ‘local‘ keyword. This prevents them from affecting variables outside the function.

```
function myfunc {
    local myvar="local variable"
    echo $myvar
}
```

4.3.5 Function Scope

In Bash, functions are executed in the context of the current shell. Therefore, changes made to variables in a function are visible outside the function.

4.4 String Manipulation

Bash provides a range of operations for manipulating strings, which is essential for text processing and script automation.

4.4.1 String Length

To find the length of a string:

```
#!/bin/bash
str="Bash Programming"
echo ${#str}  % Outputs the length of str
```

4.4.2 Substring Extraction

Extract a substring from a string:

```
# Extracts substring starting at position 5 with length 4
echo ${str:5:4}
```

4.4.3 Substring Replacement

Replace parts of a string:

```
# Replaces first occurrence of 'Bash' with 'Shell'
echo ${str/Bash/Shell}

# Replace all occurrences of a substring
echo ${str//a/A}
```

4.4.4 String Concatenation

Strings can be concatenated simply by placing them next to each other:

```
str1="Hello"
str2="World"
echo $str1$str2  % Outputs 'HelloWorld'
```

4.4.5 Trimming Strings

Trimming leading and trailing characters:

```
# Trim leading characters
str="     Bash"
echo ${str#* }

# Trim trailing characters
str="Bash     "
echo ${str% *}
```

4.4.6 Case Conversion

Convert string case:

```
# To uppercase
echo ${str^^}

# To lowercase
echo ${str,,}
```

4.4.7 Pattern Matching and Extraction

Extract parts of a string based on a pattern:

```
str="file-123.txt"
# Extract numbers from the string
[[ $str =~ [0-9]+ ]] && echo ${BASH_REMATCH[0]}
```

String manipulation capabilities in Bash are powerful and versatile, enabling the handling of text data in various ways, from simple modifications to complex pattern matching.

4.5 Advanced File Operations

Bash provides a wide array of capabilities for handling complex file operations, including file redirection, manipulation of file descriptors, and processing file content.

4.5.1 Redirection

Redirection is used to control where the output of a command goes or where input comes from.

Output redirection:

```
#!/bin/bash
echo "Hello World" > output.txt  % Writes output to output.txt
```

Appending to a file:

```
echo "Another line" >> output.txt  % Appends to output.txt
```

Input redirection:

```
while read line; do
    echo $line
done < input.txt  % Reads from input.txt
```

4.5.2 Pipes

Pipes are used to pass the output of one command as input to another.

```
cat input.txt | grep "search term"  % Searches input.txt for "search term"
```

4.5.3 File Descriptors

File descriptors are small, non-negative integers that identify a file to the system.

Standard file descriptors:

- 0: Standard input (stdin)
- 1: Standard output (stdout)
- 2: Standard error (stderr)

Redirecting stderr:

```
command 2> error.log  % Redirects stderr to error.log
```

4.5.4 Here Documents

A here document is used to redirect input into an interactive shell script or program.

```
cat << EOF
This is a line
Another line
EOF
```

4.5.5 Process Substitution

Process substitution lets you treat the output of a command as if it were a file.

```
diff <(command1) <(command2)  % Compares output of two commands
```

4.5.6 File Testing

Bash provides operators to test various file properties.

Check if a file exists:

```
if [ -f "$filename" ]; then
    echo "File exists."
fi
```

Check if a directory exists:

```
if [ -d "$directoryname" ]; then
    echo "Directory exists."
fi
```