Control Structures in Shell

Unit-IV

SCS281: Linux and Shell Programming

Mapped Course Outcomes (CO): CO3

Control Structures in Shell

- Control structures in shell scripting enable decision-making and looping, similar to other programming languages.
- They include **conditional statements** and **loops**.

Control Structure are:

- **Decision**: if, if-else, case.
- **Repetition**: for, while, until.
- **Exiting or Skipping**: break, continue.

Decision Making: Conditional Statements

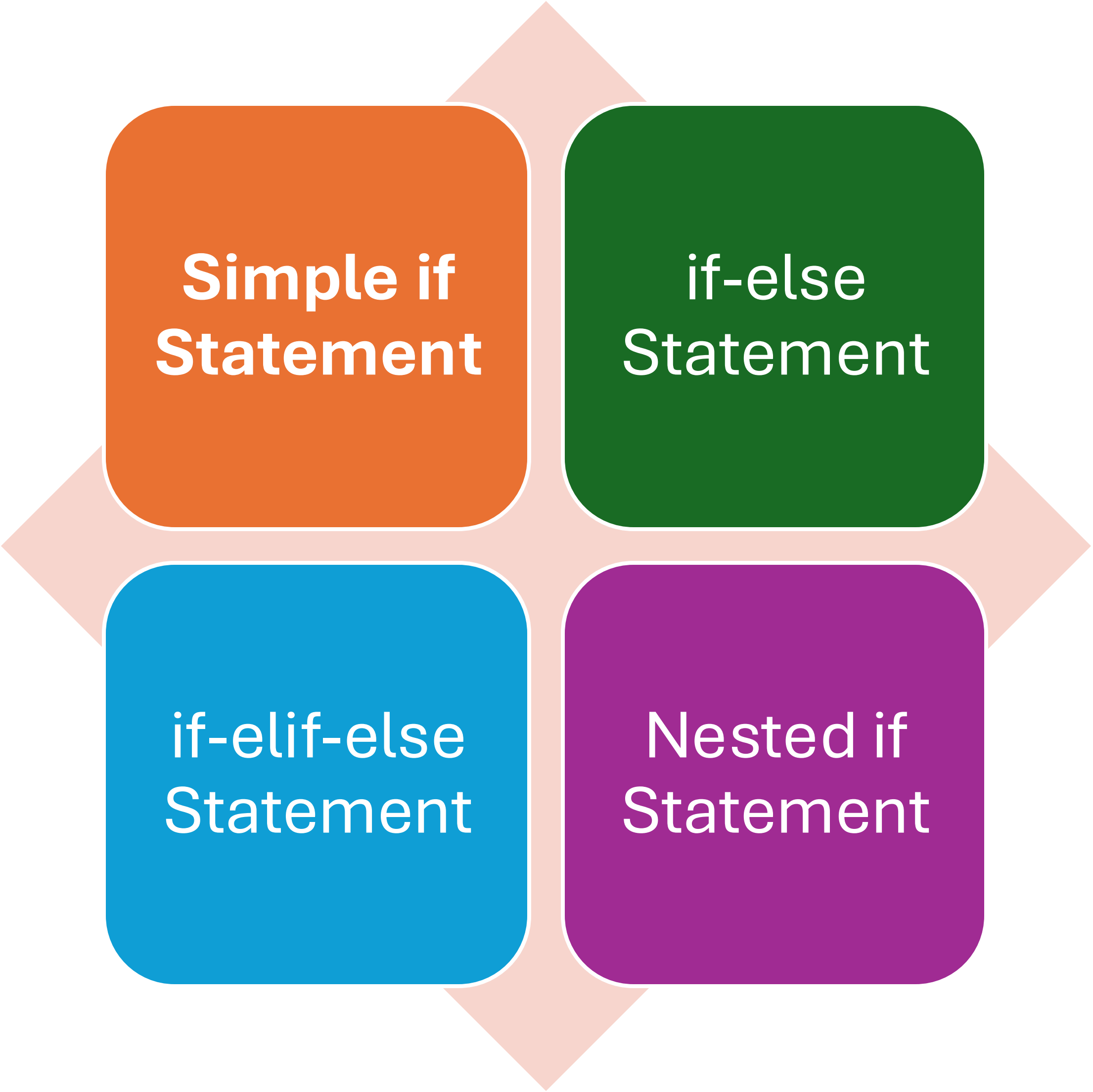
- The if statement is a fundamental control structure in Linux shell scripting.
- It is used to evaluate conditions and execute commands based on whether the condition evaluates to **true** or **false**.

Syntax

Basic if Statement

```
if [ condition ]; then
   # Commands to execute if condition is true
fi
```

Types of if Statements



Simple if Statement

if-else Statement

if-elif-else Statement

Nested if Statement

Simple if Statement

if Statement Executes a block of code if a condition is true.

Syntax:

```
if [ condition ]; then
# Commands to execute if #condition is true
fi
```

Example:

```
if [ -f "file.txt" ]; then
   echo "File exists!"
else
   echo "File does not exist."
fi
```

if-elif-else

Tests multiple conditions.

Syntax:

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

Example:

```
read -p "Enter a number: " num
if [ $num -gt 0 ]; then
  echo "Positive number"
elif [ $num -lt 0 ]; then
  echo "Negative number"
else
  echo "Zero"
fi
```

if-elif-else Statement

Allows multiple conditions to be evaluated sequentially. Executes the first matching condition's block.

Syntax

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

Example

```
#!/bin/bash
read -p "Enter a number: " num
if [ $num -gt 0 ]; then
   echo "Positive number"
elif [ $num -lt 0 ]; then
   echo "Negative number"
else
   echo "The number is zero"
fi
```

Nested if Statement

if statements can be nested for complex logical operations.

Syntax:

```
if [ condition1 ]; then
   if [ condition2 ]; then
      # Commands to execute if condition1 and condition2 are true
   fi
fi
```

Ex**ample**

```bash
#!/bin/bash
read -p "Enter your age: " age
if [ $age -ge 18 ]; then
   if [ $age -lt 60 ]; then
      echo "You are eligible to work."
   else
      echo "You are of retirement age."
   fi
else
   echo "You are not eligible to work."
fi
```

if with Logical Operators

Combines multiple conditions using logical operators like **AND** (&&), **OR** (||), and **NOT** (!).

Syntax:

```
if [ condition1 ] && [ condition2 ]; then
   # Commands if both conditions are true
fi
if [ condition1 ] || [ condition2 ]; then
   # Commands if at least one condition is true
fi
if [ ! condition ]; then
   # Commands if condition is false
fi
```

Example 1: Using AND (&&)

```
#!/bin/bash
read -p "Enter two numbers: " num1 num2
if [ $num1 -gt 10 ] && [ $num2 -lt 20 ]; then
   echo "First number > 10 AND second number < 20."
fi
```

if with Logical Operators

Example 2: Using OR (||)

```
#!/bin/bash
read -p "Enter a filename: " file
if [ -f "$file" ] || [ -d "$file" ]; then
   echo "$file exists."
else
   echo "$file does not exist."
fi
```

Example 3: Using NOT (!)

```
#!/bin/bash
read -p "Enter a filename: " file
if [ ! -f "$file" ]; then
   echo "$file is not a regular file."
fi
```

Arithmetic Comparisons

Test	Description
[num1 -eq num2]	True if equal.
[num1 -ne num2]	True if not equal.
[num1 -gt num2]	True if greater.
[num1 -lt num2]	True if less.
[num1 -ge num2]	True if greater or equal.
[num1 -le num2]	True if less or equal.

Example

```
#!/bin/bash
read -p "Enter two numbers: " a b
if [ $a -gt $b ]; then
   echo "$a is greater than $b."
else
   echo "$b is greater than or equal to $a."
fi
```

case Statement

Simplifies multi-condition checks.

Syntax:

```
case value in
  pattern1)
    # Commands for pattern1
    ;;
  pattern2)
    # Commands for pattern2
    ;;
  *)
    # Default commands
    ;;
esac
```

Key Points:

1. **expression**: The variable or command output being tested.
2. **pattern**: Specifies the condition to match. Patterns can include wildcards (*, ?, [ ]) or regular expressions.
3. **;;**: Ends the commands for a particular pattern.
4. ***)**: Represents the default case (like else in an if-else structure).
5. **Whitespace**: Each pattern must end with a closing parenthesis) and be followed by commands.

Examples

```
#!/bin/bash
read -p "Enter a day of the week: " day
case $day in
  Monday)
    echo "Start of the workweek."
    ;;
  Friday)
    echo "End of the workweek!"
    ;;
```

```
Saturday|Sunday)
    echo "It's the weekend!"
    ;;
  *)
    echo "Not a valid day."
    ;;
esac
```

Using Wildcards

```
#!/bin/bash
read -p "Enter a filename: " filename

case $filename in
   *.txt)
      echo "It's a text file."
      ;;
   *.sh)
      echo "It's a shell script."
      ;;
```

```
*.jpg|*.png)
        echo "It's an image file."
        ;;
    *)
        echo "File type unknown."
        ;;
esac
```

Using ranges

```
#!/bin/bash
read -p "Enter a single character: " char

case $char in
    [a-z])
        echo "You entered a lowercase letter."
        ;;
    [A-Z])
        echo "You entered an uppercase letter."
        ;;
 [0-9])
        echo "You entered a digit."
        ;;
    ?)
        echo "You entered a special character."
        ;;
    *)
        echo "Invalid input."
        ;;
esac
```

Case-Insensitive Matching

By default, case is case-sensitive. To make it case-insensitive, convert input to lowercase using tr or shopt.

```
#!/bin/bash
read -p "Enter a day: " day
day=$(echo $day | tr '[:upper:]' '[:lower:]') # Convert to lowercase

case $day in
    monday)
        echo "Start of the workweek."
        ;;
    friday)
        echo "End of the workweek!"
        ;;
    saturday|sunday)
        echo "It's the weekend!"
        ;;
    *)
        echo "Not a valid day."
        ;;
esac
```

Using Commands in the Expression

The case statement can evaluate the output of a command.

```
#!/bin/bash
os_type=$(uname)

case $os_type in
    Linux)
        echo "You're using a Linux system."
        ;;
    Darwin)
        echo "You're using macOS."
        ;;
    *)
        echo "Unknown operating system."
        ;;
esac
```

Menu example

```
#!/bin/bash


echo "Choose an option:"
echo "1) Show date"
echo "2) Show files"
echo "3) Exit"
read -p "Enter your choice: " choice


case $choice in
   1)
      date
      ;;
 2)
      ls
      ;;
   3)
      echo "Goodbye!"
      exit 0
      ;;
   *)
      echo "Invalid choice."
      ;;
esac
```

Comparison with if-elif-else

Feature	if-elif-else	case Statement
Syntax Complexity	Comparatively verbose	More concise for multiple cases
Pattern Matching	Limited	Supports wildcards and ranges
Readability	Harder for many conditions	Easier with many conditions
Use Case	Complex logical conditions	Menu-driven or pattern matching

Loops in Shell Script

Loops are control structures that allow the repetition of commands based on certain conditions.

Linux shell scripting supports several types of loops, including for, while, and until.

Each type of loop has its own use cases and syntax.

for Loop

The for loop iterates over a list of items, executing commands for each item in the list.

Syntax

```
for variable in list; do
   # Commands to execute
done
```

Features

- Iterates through a predefined list of values.
- Suitable for iterating over files, strings, or ranges.

Example-Iterate Over a List

Example

```
#!/bin/bash
for color in red green blue; do
   echo "The color is $color"
done
```

Output

```
The color is red
The color is green
The color is blue
```

Example-Numeric Range with { }

Example

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

Output

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

while Loop

The while loop executes commands as long as the specified condition is true.

Syntax

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

Example

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

Output

```
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
```

until Loop

The until loop is the opposite of the while loop. It executes commands as long as the condition is **false**.

Syntax

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

Example

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

Output

```
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
```

Nested Loops

Loops can be nested to handle more complex tasks.

Example

```
#!/bin/bash
for i in {1..3}; do
  for j in {1..2}; do
    echo "Outer: $i, Inner: $j"
  done
done
```

Output

```
Outer: 1, Inner: 1
Outer: 1, Inner: 2
Outer: 2, Inner: 1
Outer: 2, Inner: 2
Outer: 3, Inner: 1
Outer: 3, Inner: 2
```

Controlling Loops

1. break Statement

Exits the loop immediately.

Example:

```
#!/bin/bash
for i in {1..5}; do
   if [ $i -eq 3 ]; then
      break
   fi
   echo "Number: $i"
done
```

Output

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

Controlling Loops

2. continue Statement

Skips the current iteration and proceeds to the next.

Example:

```
#!/bin/bash
for i in {1..5}; do
   if [ $i -eq 3 ]; then
      continue
   fi
   echo "Number: $i"
done
```

Output

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

Loop with Command Outputs

Loops can iterate over the output of commands.

Example:

```
#!/bin/bash
for user in $(cat /etc/passwd | cut -d: -f1); do
   echo "User: $user"
done
```

Explanation:

cat /etc/passwd | cut -d: -f1:

Lists all users from /etc/passwd.

Comparison of Loops

Feature	for Loop	while Loop	until Loop
Use Case	Iterating over a predefined list	Executes while condition is true	Executes until condition is true
Syntax	Compact	Suitable for dynamic conditions	Similar to while, but condition is inverted
Termination	When all items are processed	When condition becomes false	When condition becomes true