System Programming - Linux
Week 4 - Control Structures

Lecturer: Dr. Aggrey Obbo (PhD)

April 22, 2025

Contents

1 Introduction

A control structure is a basic principle in programming that dictates how the execution of a program proceeds. It enables a program to make choices, repeat actions, and perform particular tasks based on certain conditions or defined criteria. Typical examples of control structures include conditional statements (such as if and else), loops (like for and while), and function invocations. Understanding its constructs is essential for automating system administration, processing data, and executing complex command sequences in Linux environments.

Objectives

- Explain concepts and terminologies in control structures
- Create conditional statements: Write scripts using if, else, and elif to run commands depending on defined conditions.
- Build iterative loops: Develop scripts that use for, while, or until loops for tasks that need to be repeated.
- Use case statements: Implement case statements for effective multi-way decision-making based on pattern recognition.
- Examine script execution: Track and anticipate the flow of execution in scripts that include various control structures.

2 Concepts and Terminologies in Control Structures

2.1 The Concept

Consider traffic lights as a form of control mechanism. The initial condition is a vehicle nearing the intersection. This prompts the light's system into action. The system then assesses the current status of the light.

When the light is green, the resulting action is the vehicle moving forward or turning, as permitted. And When the light is yellow, the outcome depends on the vehicle either moving cautiously if stopping is not safe or coming to a stop before reaching the intersection.

When the light is red, the expected action is the vehicle coming to a complete halt. The control mechanism (traffic light system) utilizes the initial condition (approaching vehicle) to yield different outcomes (vehicle movement) depending on the light's status.

The arriving moving car is the precondition, light's status serves as the criterion that dictates the resultant action/postcondition (car continuing or stopping). This illustration demonstrates how an initial condition initiates a control mechanism, resulting in various outcomes based on assessed conditions.

In the realm of Linux, especially within shell scripting (such as Bash), control structures are programming elements that dictate the order in which commands are executed. They enable you to:

- Make choices: This means executing different groups of commands depending on whether specific conditions are met or not. Examples include if, else, elif, and case statements.
- Repeat tasks: This refers to running a set of commands several times, either for a specified number of times or until a particular condition is satisfied. Examples include for, while, and until loops.

In essence, control structures empower your shell scripts to carry out intricate tasks by incorporating logic and repetition. They are essential for automation and for developing versatile scripts.

2.2 Terminologies

Control structures, the backbone of program flow, employ different terminologies. Below are some of the terminologies in control structures [1][2]:

- **Selection**

  This refers to the process of picking a code path based on certain conditions. "If-else" statements illustrate selection by guiding the flow of the program according to the truth value of a condition.

- **Condition**

  A boolean statement that can be either true or false. Conditions govern the control flow in selection and looping structures, serving as the foundation for decision-making in a program.

  A precondition refers to a requirement that must be fulfilled before a particular code segment or function can run. While a postcondition outlines the state that should hold true once a code segment or function has finished running. It describes the anticipated results or impacts of the execution. Postconditions act as a assurance, verifying that the code achieved the desired outcome and preserved data integrity.

  It describes the anticipated condition of the program or inputs, ensuring that the code functions properly. Preconditions serve as protective measures, avoiding errors by confirming that essential conditions are satisfied.

- Iteration

  Iteration, often referred to as looping, involves executing a section of code repeatedly until a specified condition is satisfied. Common structures for this purpose include "for" and "while" loops, which facilitate the execution of repetitive tasks.

- **Branching**

  A form of selection where program execution diverges into different paths based on evaluated conditions, like a "switch" or "case" statement.

- **Sequence**

  This refers to the linear execution of code statements, one after another, in the order they appear. It's the fundamental flow in most programs

Control statements are those that alter the sequence in which statements are executed. Examples include If, If-else, Switch-Case, and while-do statements. Control statements have been categorized as follows[1][2][3]:

3 Conditional Statements

These are statements that will be executed when a condition specified is true. They enable decision-making, altering script flow. In Linux scripting, conditional statements play a vital role in allowing scripts to be flexible and interactive. The if statement is the most basic, enabling the script to run different sections of code depending on whether a specified condition is true or false. Its format is:

If [condition]; then [command] fi

The [condition] section assesses an expression, which could involve comparing variables or verifying the existence of files. If the condition holds true, the commands between then and fi are executed. The logic can be further extended using elif and else.

Example

```
#!/bin/sh

    a=5
    b=30

    if [ a <b ]; then
    echo "a is less than b"
    fi
```

The script prints out "a is less than b" because the value of a is less than the value of b.

4 Iterative Loops

Iterative loops serve as essential control structures in shell scripting, allowing commands to be executed repeatedly. They automate tasks by carrying out actions multiple times, which is vital for system administration, data processing, and automation. There are three main types of loops: for, while, and until. The for loop processes a defined list, while loops run as long as a specified condition holds true, and until loops function until a particular condition is met. These loops, along with the select loop for creating menu-driven scripts, simplify tasks that would typically need to be performed manually, thereby improving efficiency and automation in Linux settings.

4.1 For Loop

The For Loop iterates over a specified list or range. It's meant for tasks that require a fixed number of iterations, such as handling lists of files, going through command-line arguments, or executing operations on a sequence of numbers. Its format is well-suited for scenarios where the extent of the loop is established in advance.

The the syntax is as follows:

```
For var in firstvalue secondvalue ...n^thvalue
do
    command to be executed for every word.
done
```

Example

```
#!/bin/sh
echo "Numbers to find their square:"
for i in 1..3; do
    square=$((i * i)) # Calculate the square
    echo "Number: $i, Square: $square"
done
```

Explanation: for i in 1..3; do ... done iterates through numbers 1 to 3. square=$((i * i)) calculates the square of each number. The echo prints the number and its square.

4.2 While Loop

A while loop serves as a basic control flow structure in programming, allowing a block of code to run repeatedly as long as a defined condition holds true. The while loop operates by continually executing a block of code while a specified condition is satisfied. It checks the condition before each repetition, runs the code block if the condition is true, and stops executing when the condition turns false. This mechanism enables dynamic iteration based on evolving conditions within a program.

The syntax of a While Loop in bash scripting is as follows:

```
while [ condition ];
do
    # code block
done
```

Example

```
#!/usr/bin/bash
a=5
while [ $a > 3 ];
do
    echo $a
    ((a–))
done
echo “Out of the loop”
```

Explanation

#!usr/bin/bash # Shell path
a=5 # Initializing the variable.
while [$a > 3]; # Starts the while loop.
do # Beginning of the code block to be executed within the while loop.
echo $a # Prints the current value of the variable
((a–)) # Decrements the value of a by 1. To ensure repetition for multiple values
done # Marks the end of the code block for the while loop.
echo ”Out of the loop” # Prints “Out of the loop”

4.3 Until Loop

An ”until” loop repeatedly executes a block of code until a particular condition is satisfied. After every iteration, the loop evaluates the condition. If the condition is false, the loop persists; when it becomes true, the loop ends. This is different from ”while” loops, which operate as long as a condition remains true. It is useful for tasks that require waiting for a specific event to occur, such as waiting for a file to be created.

The basic until loop in bash is as follows:

```
until [Condition]
do
    # Commands
done
```

Explanation

until - Creates a new until loop.
condition - Specifies the statement to be executed when the condition is false.
do - Specify the start of the loop.
done - Marks end of the loop.

Example

```
#!/bin/bash
count=1
until [ "$count" = 5 ]; do
    echo "The current count is: $count"
    ((count++))
done
```

The output to the above script should give a count up to 5.

4.4 Select Loop

This loop presents a menu of options that will allow an interactive selection to the user. It streamlines the development of scripts with menus, facilitating the management of user input and the execution of related actions. It is crafted for user engagement, offering an organized approach to display options and react to user choices.

The syntax of a bash select loop is as follows;

```
select var in element1 element2 ... elementn
do
    Statement to be executed for every element.
done
```

Explanation

Variable var is the name of a variable with elements element1, element2, ...elementn. Each time the for loop executes, the value of the variable var is set to the next word in the list of elements.

Example

```
#!/bin/ksh
    select Car_Brand brand in Corona Corona Corolla Lexus Hino all non
do
    case $Car_Brand in
        Rav4 |Corona |Corolla |all)
        echo “Go to car bond”
        ;;
        Lexus |Hino)
        echo “Already have them”
        ;;
        none)
        break
        ;;
        *) echo “ERROR: Invalid selection”
        ;;
    esac
done
```

The resulting menu lists all the options indicating which options may already exist.

5 Case Statements

In Bash scripting, the case statement serves as a control flow mechanism that enables the execution of different code blocks depending on the value of a variable or expression. It functions similarly to the switch statement found in other programming languages. The syntax consists of the “case” keyword followed by the value to be matched, the “in” keyword, and one or more patterns with corresponding code blocks enclosed in “;;” statements as follows:

```
case EXPRESSION in
    pattern_1)
        statements
        ;;

    pattern_2)
        statements
        ;;

    pattern_n)
        statements
        ;;

    *)
        statements
        ;;
esac
```

Explanation

The case statement starts with a case keyword, followed by case expression and the in keyword. It then ends with the esac keyword. It then follows with multiple patterns separated by the — operator. The “)” operator terminates a pattern list. A pattern can have special characters. The pattern some times consisting of special characters are terminated with “;;”. The commands corresponding with the first pattern that matches the expression are executed. And the wildcard asterisk symbol (*) as a final pattern to define the default case. It will always match if no other pattern is matched, and the return status is normally zero. Otherwise, the return status is the exit status of the executed commands. The “esac” characters signify the end of the case statements.

Example

```
#!/bin/bash
echo -n “Enter an East African country to determine its official language: ”
read country
echo -n “The official language of $country is ”
case $COUNTRY in
    Uganda)
        echo -n “English”
        ;;
    Kenya |Tanzania)
        echo -n “Swahili”
        ;;
    *)
        echo -n “This country is not found in East Africa”
        ;;
esac
```

The script will ask you to enter a country. If you type “Uganda”, it will match the first pattern, and the echo command will print out ”The official language of Uganda is English. For Kenya or Tanzania, it will match the second pattern and print out “The official language of Kenya or Tanzania is Swahili”.

6 Examining Script Execution

Verifying the correct execution of a shell script is an essential step in the development process. Here are some techniques you can use to ensure your script is working as expected[3]:

- Testing: Cases can also be created to test and verify the functionality of the script under various conditions.
- Output Inspection: Analyze the output of your script to ensure it’s producing the expected results (echo commands can be used).
- Analyze the exit Codes: Shell scripts return an exit code upon completion. A non-zero exit code typically indicates an error. The exit code can be checked using the $? shell variable.

  ```
  #!/bin/bash
  set -x
  value=20
  if [[ $value > 10 ]]; then
  echo “Value ($value) is greater than 10”
  result=$(($value * 2))
  echo ”Result: $result”
  else
  echo “Value ($value) is not greater than 10”
  fi
  ```

  In this case, the “echo” reveals the branch taken and variable values, clarifying the if condition’s outcome.
- Debugging: The set -x command can enable shell script debugging. It displays each command as it’s executed, helping one identify the source of any issues.
- Error Handling: Error handling can be implemented in the script using conditional statements, such as if-then-else blocks, to handle different scenarios and provide meaningful error messages. For example:

In conclusion, gaining expertise in Bash control structures is crucial for creating resilient and flexible scripts. Conditional statements (if, elif, else) support decision-making processes, while loops (for, while, until) help manage repetitive tasks. Merging these elements permits the creation of intricate logic, which is vital for automating system administration, data processing, and more. Skillful utilization of these tools, along with strong debugging methods, will greatly improve one’s scripting skills.

List of References

1. Cooper, M. (2014). Advanced Bash scripting guide. .
2. Ramey, C. (1998). Bash reference manual. Network Theory Limited, 15.
3. Blum, R. (2008). Linux command line and shell scripting bible (Vol. 481). John Wiley Sons.