20CYS312 - Principles of Programming Languages
Exploring Programming Paradigms

Assignment-01

Presented by Siddharth Krishna
«CB.EN.U4CYS21058»
TIFAC-CORE in Cyber Security
Amrita Vishwa Vidyapeetham, Coimbatore Campus

Feb 2024

Outline

Scripting

What is Scripting:
Like all programming, scripting is a way of providing instructions to a computer so you can tell it what to do and when to do it. Programs can be designed to be interacted with manually by a user (by clicking buttons in the GUI or entering commands via the command prompt) or programmatically using other programs (or a mixture of both).

Consider this web page displayed on your browser – the browser is a program that you can interact with manually, and that program also reads other code (the HTML and CSS that describes the page) to determine what to display.

Scripting Languages: Javascript, PHP, Python, Perl, Bash

BASH

Bash (Bourne Again SHell) is a command processor that typically runs in a text window where the user types commands that cause actions. It is the default shell on most Linux distributions and macOS. Bash is primarily associated with the imperative and procedural programming paradigms.
Bash scripting supports variables, conditional statements, and loops just like programming languages. Bash scripting is a great way to automate different types of tasks in a system

Applications of Bash Scripts:

- Manipulating files
- Executing routine tasks like Backup operation
- Automation

Scripting - Bash

Script structure is crucial for creating readable, maintainable, and efficient Bash scripts.
Script Structure:

1. Bash scripts are typically a series of commands written in a text file with a .sh extension.

2. The script is executed sequentially, line by line, from top to bottom.

3. The shebang (!) at the beginning of a script specifies the interpreter to use. eg. !/bin/bash

4. Comments and Documentation:
Include comments to explain the purpose of the script, author information, and any relevant details.
eg. #!/bin/bash
# Script: my_script.sh
# Author: Your Name
# Description: This script does XYZ

Scripting - Bash

5. Global Variables:
Declare global variables that will be used throughout the script. eg.
#Global variables
MY_VARIABLE="some_value"

6. Functions:
Define functions to modularize your code and improve readability.
eg. # Function to perform a specific task
function my_function() {
function code
}

7. Main Execution:
Include the main execution logic of your script. This section might involve calling functions, conditional statements, and loops.
eg.
Main execution
echo "Starting script..."
$my_function$
$echo"Scriptcompleted."$

Scripting - Bash

8. Command-Line Arguments:
If your script accepts command-line arguments, handle them in a dedicated section.
eg.

```
# Command-line argument handling
if [[ $# -eq 0 ]]; then
echo "Usage: $0 <argument>"
exit 1
fi
```

9. Error Handling:
Implement error handling to gracefully manage unexpected situations.

```
# Error handling
function handle_error() {
echo "Error: $1"
exit 1
}

# Example usage
command_that_might_fail || handle_error "Command failed."
```

Scripting - Bash

10. Cleanup:
Include cleanup logic if your script uses temporary files or resources.

```
# Cleanup
function cleanup() {
cleanup code
}
# Trap signals for cleanup
trap cleanup EXIT
```

11. Exit Codes:
Use exit codes to indicate the success or failure of your script.

```
# Exit codes
exit_code=0

# Main execution
if [ some_condition ]; then
# code
else
exit_code=1
fi
exit $exit_code
```

Imperative Programming

Imperative programming is a software development paradigm where Functions are implicitly coded in every step required to solve a problem. In imperative programming, every operation is coded and the code itself specifies how the problem is to be solved, which means that pre-coded models are not called on.

Imperative vs Declarative Programming

Imperative programming contrasts with declarative programming, in which how a problem is solved is not specifically defined, but instead focuses on what needs to be solved. Declarative programming provides a constant to check to ensure the problem is solved correctly, but does not provide instructions on how to solve the problem.
The exact manner in which the problem is solved is defined by the programming language's implementation through models. Declarative programming is also called model-based programming.
Functional, domain-specific (DSL) and logical programming languages fit under declarative programming, such as SQL, HTML, XML and CSS.

The models from which declarative programming gets its functions are created through imperative programming. As better methods for functions are found through imperative programming, they can be packaged into models to be called upon by declarative programming.

Imperative - Rust

Rust is a multi-paradigm systems programming language that supports both imperative and functional programming styles.

Imperative - Rust

1. Variable Declaration:
In Rust, variables are immutable by default. You can use the let keyword to declare variables and the mut keyword to make them mutable.

```
let x = 10; // Immutable variable
let mut y = 5; // Mutable variable
```

2. Control Flow:
Rust supports if, else, else if, match, while, and for for controlling the flow of execution.

```
// Example of if-else statement
if x > 5 {
println!("x is greater than 5");
} else {
println!("x is not greater than 5");
}
```

3. Functions:
You can define functions in Rust using the fn keyword. Functions can have parameters and return values.

```
fn add(a: i32, b: i32) -> i32 {
a + b
}
```

Imperative - Rust

4. Mutable References:
Rust has a unique ownership system, and mutable references ensure that only one part of the code can modify a particular piece of data at a time.

```
fn main()
let mut value = 42;
modify_value(mut value);
println!("Modified value: ", value);
fn modify_value(x: mut i32) {
*x += 10;
}
```

5. Loops:
Rust supports loop, while, and for loops for iteration.

```
// Example of a for loop
for i in 0..5 {
println!("Iteration ", i);
}
```

Imperative - Rust

6. Pattern Matching:
Rust's powerful pattern matching feature, known as match, allows for complex control flow based on pattern matching.

```
let number = 42;
match number {
0 => println!("Zero"),
1 | 2 => println!("One or Two"),
_ => println!("Other"),
}
```

7. Error Handling:
Rust encourages the use of the Result type for error handling, ensuring that errors are explicitly handled.

```
fn division(a: i32, b: i32) -> Result<i32, 'static str> {
if b == 0 {
Err("Division by zero")
} else {
Ok(a / b)
}
}
```

Imperative - Rust

8. Structs and Enums:
Rust allows you to define custom data structures using struct and enum, enabling you to encapsulate data and behavior.
struct Point {
x: i32,
y: i32
}
enum Direction {
Up,
Down,
Left,
Right
}

9. Owernship:
Rust's ownership system, with concepts like borrowing and lifetimes, ensures memory safety without garbage collection.

10. Closures:
Rust supports closures, which are similar to anonymous functions, allowing you to capture variables from their surrounding environment.

Comparisons

Scripting - Bash	Imperative - Rust
1. Use Case:Primarily used for automating system tasks, quick scripting, and command-line utilities.	A systems programming language designed for performance, memory safety, and low-level control.
2. Domain:Well-suited for tasks like file manipulation, text processing, and system administration.	Used for building applications, libraries, and services where performance and control over system resources are critical.
3. Syntax: Simple and concise syntax suitable for quick scripts and automation tasks.	More expressive syntax with a focus on readability and maintainability.
4. Control Flow: Limited control flow constructs compared to Rust. if, else, for, and while loops are available for basic control structures.	Rich set of control flow constructs, including if, else, while, for, and powerful pattern matching with match. Provides more expressive control flow for complex logic.

Comparisons

Scripting - Bash	Imperative - Rust
5. Memory Management: Memory management is handled automatically by the shell, and users typically don't need to manage memory explicitly.	Employs a unique ownership system for memory safety without garbage collection. Requires explicit handling of memory through concepts like borrowing and lifetimes.
6. Concurrency and Parallelism: Limited support for concurrency and parallelism. Often relies on external utilities for parallel execution.	Strong support for concurrency and parallelism with features like threads, channels, and the ownership system
7. Error Handling: Relies on exit codes and conditional checks for error handling.	Forces developers to handle errors, reducing the chance of unchecked runtime errors.
8. Ecosystem and Libraries: Strong integration with Unix/Linux utilities and tools. Limited native support for complex data structures and algorithms	Growing ecosystem with a focus on performance and safety. Rich set of libraries and frameworks for various domains, including networking, systems programming, and web development.

References

1. geeksforgeeks
2. github
3. RUST
4. TECHTARGET
5. Youtube
6. courseera
7. stackoverflow
8. chatgpt
9. Javaatpoint