🧮 Bash Calculator: Functions, Input Handling & Error Control

📌 Overview

This project demonstrates the use of **Bash functions, user input handling, conditional logic, and error checking** through a simple four-function calculator.

While the calculator itself is basic, the scripting patterns used here are foundational for **security automation, administrative tooling, and defensive scripting**, where reliability and input validation matter more than complexity.

🎯 Objective

- Define and use multiple Bash functions
- Accept and process user input from the command line
- Perform arithmetic operations safely
- Handle error conditions gracefully (e.g., division by zero)
- Control program flow using a `case` statement

🧩 Scenario

In cybersecurity and system administration, scripts frequently:

- Accept user input or operator commands
- Route logic based on user choice

- Perform calculations (thresholds, counters, limits)
- Fail safely when invalid input is provided

This calculator models those same patterns in a controlled, easy-to-understand context.

🛠️ Tools & Technologies

- Bash shell scripting
- Functions with parameters
- Arithmetic expansion
- Conditional statements
- `case` control flow
- Standard input (`read`)
- Error messaging

🔍 Methodology

1️⃣ Function-Based Design

- Implemented separate functions for:
    - Addition
    - Subtraction
    - Multiplication
    - Division

- Each function:
    - Accepts two parameters
    - Performs a single, well-defined task
    - Returns output via `echo`

Security relevance:
Function separation improves readability, testability, and reduces logic errors in automation scripts.

2️⃣ Input Collection & Validation

- Collected numeric input interactively
- Required explicit operator selection
- Used a `case` statement to restrict valid operations

Security relevance:
Restricting valid inputs reduces unexpected behavior and logic misuse in operational scripts.

3️⃣ Error Handling (Division by Zero)

- Explicitly checked for division by zero
- Returned a clear, user-friendly error message instead of failing silently

Security relevance:
Fail-safe behavior is critical in scripts that may be reused during investigations or operational tasks.

4️⃣ Controlled Execution Flow

- Used quoted operators in the `case` statement to avoid shell wildcard expansion
- Ensured only the intended branch executes

Security relevance:
This demonstrates awareness of shell parsing behavior and defensive scripting practices.

▶️ Example Usage

```bash
#!/bin/bash

#take inputs from a user and add numbers
add() {
  echo $(($1 + $2))
}

#take inputs from a user and subtract numbers
subtract() {
  echo $(($1 - $2))
}

#take inputs from a user and multiply numbers
multiply() {
  echo $(($1 * $2))
}

#take inputs from a user and devide numbers
#display an error message if the division by zero
divide() {
  if [ $2 -eq 0 ]; then
    echo "Error: Division by zero is not allowed."
  else
    echo $(($1 / $2))
  fi
}
```

```
echo "Enter first number: "
read num1
echo "Enter second number: "
read num2
echo "Enter operation (+, -, *, /): "
read op

case $op in
  "+") result=$(add $num1 $num2) ;;
  "-") result=$(subtract $num1 $num2) ;;
  "*") result=$(multiply $num1 $num2) ;;
  "/") result=$(divide $num1 $num2) ;;
  *) result="Error: Invalid operation" ;;
esac

echo "Result: $result"
```

```
$ ./calculator.sh
Enter first number: 10
Enter second number: 5
Enter operation (+, -, *, /): +
Result: 15

$ ./calculator.sh
Enter first number: 10
Enter second number: 5
Enter operation (+, -, *, /): -
Result: 5

$ ./calculator.sh
Enter first number: 10
Enter second number: 5
Enter operation (+, -, *, /): *
Result: 50
```

🧠 Key Takeaways

- Bash functions enable modular, reusable logic

- Input validation and error handling are essential even in small scripts
- Defensive scripting habits scale directly to security automation