Command Line Calculator

Mahmut Doğukan Kara

ID : B2305.090233

github: github.com/Dogude/Command-Line-Calculator

Problem statement

In today’s fast-paced world, users often need a simple and lightweight tool for performing basic mathematical calculations without the overhead of using complex software or navigating online tools. Many available solutions are either overly complicated or require internet connectivity, which is not always accessible.

The problem is the lack of a reliable, quick, and easy-to-use command-line calculator that operates offline and provides essential mathematical operations with robust error handling.

Problem statement

To develop a **Command Line Calculator**, a lightweight and efficient software application, that allows users to perform basic arithmetic operations (addition, subtraction, multiplication, division) in a user-friendly command-line interface. The application will focus on:

- **Ease of Use:** Simple text-based UI with clear instructions.
- **Reliability:** Robust error handling, such as preventing division by zero and handling invalid inputs.
- **Accessibility:** Available offline and operable on any terminal supporting the chosen programming language.
- **Extensibility:** Modular code structure to facilitate the addition of advanced features (e.g., modulus, power operations) in the future.
- **Educational Value:** Provide beginner programmers an example of a structured, modular, and well-documented software project.

Problem-solving approach

To create a command-line application capable of performing basic arithmetic operations (addition, subtraction, multiplication, division).

The application should handle invalid inputs and edge cases like division by zero.

Divide the functionality into smaller tasks:

- Input handling: Accept user choices and numbers.
- Arithmetic operations: Implement addition, subtraction, multiplication, and division.
- Output handling: Display results and error messages.

Consider future requirements: Make the design modular for adding advanced features later.

Design the Solution

Use a **menu-driven approach** to interact with the user.

Write modular functions for each operation (eg add, subtract).

Implement error handling for invalid inputs and edge cases.

Personas

Name: Alex, a beginner programmer learning arithmetic operations through a CLI tool.

Goals: To practice basic programming concepts and enhance debugging skills.

Name: Sam, a math enthusiast who uses the calculator for quick computations.

Goals: To practice to learn four basic arithmetic operations.

Requirements Analysis

Functional Requirements:

- Perform basic arithmetic operations (addition, subtraction, multiplication, division).
- Handle invalid inputs (e.g prevent division by zero).

Non-Functional Requirements:

- Ensure the program runs smoothly on any standard terminal.
- Maintain code readability for educational purposes.

USER STORIES

- As a user, I want to see meaningful error messages so that I can understand my mistakes and correct them.
- As a beginner programmer, I want to perform basic operations in the terminal so that I can learn arithmetic programming.
- As a developer, I want the code to be modular so that I can easily add new operations in the future.

SYSTEM DESIGN

Architecture:

- Modular functions for each operation (to be implemented in our code).
- A main menu for user interaction.

Enhancements:

- Use a loop to keep the calculator running until the user decides to exit.
- Add more operations like modulus, power, and square root.

SYSTEM DESIGN

Flowchart Example:

- Display menu.
- Accept user choice.
- Call the corresponding function.
- Show result or error message.
- Repeat or exit.

Testing Documentation

When user enters two valid numbers:

```
Calculator Menu:
1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)
Enter your choice: 1
Enter two numbers to add: 23 34
Result: 57
```

Testing Documentation

When user enters invalid number such as zero for division:

```
Calculator Menu:
1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)
Enter your choice: 4
Enter two numbers to divide: 1 0
ERROR!
Error: Division by zero is not allowed.
```

Product Backlog

As a user, I want to perform basic operations (addition, subtraction, multiplication, division) so that I can quickly calculate results.

As a user, I want error handling for division by zero so that I avoid runtime errors.

As a developer, I want modular functions so that I can add new features easily in the future.

As a user, I want clear menu instructions so that I know how to interact with the application.

As a user, I want the ability to perform advanced operations like power and modulus in future iterations.

Sprint Planning

Sprint 1 (2 weeks):

Create the CLI structure with a menu.

Implement addition, subtraction, multiplication, and division operations.

Add error handling for division by zero.

Write basic test cases for implemented operations.

Sprint Goal: Deliver a working calculator with basic operations and error handling.

Sprint Planning

-Create the CLI structure with a menu.

```cpp
int main() {
    int choice;

    std::cout << "Calculator Menu:\n";
    std::cout << "1. Addition (+)\n";
    std::cout << "2. Subtraction (-)\n";
    std::cout << "3. Multiplication (*)\n";
    std::cout << "4. Division (/)\n";
    std::cout << "Enter your choice: ";
    std::cin >> choice;

    switch(choice) {
        case 1:
            add();
            break;
        case 2:
            subtract();
            break;
        case 3:
            multiply();
            break;
        case 4:
            divide();
            break;
        default:
            std::cout << "Invalid choice.\n";
            break;
    }

    return 0;
}
```

Daily Scrum

What did you do yesterday?

- Completed the addition and subtraction functions.

What will you do today?

- Implement multiplication and division functions.

Any blockers?

- Deciding how to display error messages clearly.

Sprint Retrospective

What went well:

- ❖ Clear task division helped in completing the sprint on time.
- ❖ Error handling was implemented effectively.

What didn’t go well:

- ★ Too much time was spent debugging minor issues such as handling floating point numbers.

Improvements:

- ❖ Allocate more time for testing and debugging in the next sprint.

Adding Other Operations

add and subtract

```
void add() {
    double num1, num2;
    std::cout << "Enter two numbers to add: ";
    std::cin >> num1 >> num2;
    std::cout << "Result: " << num1 + num2 << "\n";
}

void subtract() {
    double num1, num2;
    std::cout << "Enter two numbers to subtract: ";
    std::cin >> num1 >> num2;
    std::cout << "Result: " << num1 - num2 << "\n";
}
```

Adding Other Operations

division and multiplication:

```
void multiply() {
    double num1, num2;
    std::cout << "Enter two numbers to multiply: ";
    std::cin >> num1 >> num2;
    std::cout << "Result: " << num1 * num2 << "\n";
}

void divide() {
    double num1, num2;
    std::cout << "Enter two numbers to divide: ";
    std::cin >> num1 >> num2;
    if (num2 != 0) {
        std::cout << "Result: " << num1 / num2 << "\n";
    } else {
        std::cout << "Error: Division by zero is not allowed.\n";
    }
}
```

Lessons Learned

Modularity is Key: Breaking the project into modular functions (e.g separate functions for addition, subtraction, etc) made the code easier to understand, debug, and extend.

Error Handling: Implementing robust error handling (e.g checking for division by zero) improved the user experience and made the application more reliable.

Importance of User Feedback: Iterating on the project based on user feedback (e.g adding a loop to allow multiple calculations without restarting) helped create a more user-friendly tool.

Code Documentation: Writing clear comments and a README file not only helped during development but also made the project easier for others to use and contribute to.

Agile-Scrum Methodology Benefits: Using an iterative development approach helped prioritize tasks and deliver functional increments quickly.

Future Improvements

Advanced Features:

Add more operations such as modulus, power, square root, and logarithm.

Implement support for more complex mathematical expressions (e.g 2 + 3 * 4).

Persistence:

Add a feature to log calculation history to a text file or database for later review.

Enable users to export their calculation history.

User Interface:

Add color-coded outputs in the CLI for better readability (e.g green for results, red for errors).