MCSL – 217
SOFTWARE ENGINEERING

1. **Suppose that you need to build software for a Railway Reservation System. Write a statement of scope that describes the software.**

===

Statement of Scope: Railway Reservation System Software

Purpose:

- Develop software to manage railway ticket bookings, cancellations, and modifications.
- Provide an intuitive user interface for passengers and administrative tools for railway staff.

Objectives:

- **User-Friendly Interface:** Enable easy booking, modification, and cancellation of tickets.
- **Real-Time Availability:** Show real-time seat availability and train schedules.
- **Secure Transactions:** Ensure secure payment processing and data protection.
- **Efficient Administration:** Allow railway staff to manage schedules and reservations effectively.
- **Comprehensive Reporting:** Generate reports on bookings, cancellations, and revenue.

Key Features:

1. **Passenger Module:**
    - Registration and Login
    - Train Search and Booking
    - Reservation Modification and Cancellation
    - Payment Integration
2. **Admin Module:**
    - Train Schedule Management
    - Reservation Management
    - Reporting and Analytics
3. **Security:**
    - Data Encryption
    - User Authentication
4. **Notifications:**
    - SMS and Email Alerts

Assumptions and Dependencies:

- Reliable internet connectivity.
- Integration with secure payment gateways.
- Compliance with railway regulations.

Project Timeline:

1. **Requirements Gathering:** 1 month
2. **Design and Development:** 3 months
3. **Testing and Quality Assurance:** 2 months
4. **Deployment and User Training:** 1 month
5. **Maintenance and Support:** Ongoing

2. Estimate the effort and cost required to build the above software. Use any estimation technique.

===

To estimate the effort and cost required to build the Railway Reservation System, we can use the **Function Point Analysis (FPA)** technique. FPA is a standard method for measuring the functionality delivered by the system, which can then be used to estimate effort and cost.

Step-by-Step Estimation Using FPA

1. Identify and Count Function Points:

We will classify the system's functions into different types and estimate the number of function points (FPs) for each type.

- **External Inputs (EI):** Data input by the user (e.g., registration, login, booking details).
- **External Outputs (EO):** Data output provided to the user (e.g., booking confirmation, schedules).
- **External Inquiries (EQ):** Requests with input and output (e.g., search for trains).
- **Internal Logical Files (ILF):** Databases maintained by the system (e.g., user details, booking records).
- **External Interface Files (EIF):** Interfaces to other systems (e.g., payment gateway).

Function Type	Count	Complexity	Function Points
External Inputs (EI)	10	Medium	4 x 10 = 40
External Outputs (EO)	5	Medium	5 x 5 = 25
External Inquiries (EQ)	8	Medium	4 x 8 = 32
Internal Logical Files (ILF)	4	Medium	7 x 4 = 28
External Interface Files (EIF)	2	Medium	5 x 2 = 10

Total Function Points = 40 + 25 + 32 + 28 + 10 = 135

2. Calculate Effort:

To calculate the effort, we will use an average productivity rate. Let's assume that an average developer can deliver 10 function points per person-month.

Effort = Total Function Points / Productivity Rate Effort = 135 FPs / 10 FPs per person-month Effort = 13.5 person-months

3. Estimate Cost:

To estimate the cost, we will use the average cost per person-month. Let's assume an average cost of $6,000 per person-month.

Cost = Effort x Cost per person-month Cost = 13.5 person-months x $6,000 per person-month Cost = $81,000

3. Develop SRS (Software Requirements Specification) for the Railway Reservation System (RRS)

===

Software Requirements Specification (SRS) for Railway Reservation System (RRS)

1. Introduction

1.1 Purpose

- Define functional and non-functional requirements for the Railway Reservation System (RRS).

1.2 Scope

- Manage railway ticket bookings, cancellations, modifications, and provide administrative tools for railway staff.

2. Overall Description

2.1 Product Functions

- User Registration and Login
- Train Search and Availability Check
- Ticket Booking and Payment Processing
- Reservation Modification and Cancellation
- Admin Management of Trains and Schedules
- Reporting and Analytics
- Notification via SMS and Email

2.2 User Classes and Characteristics

- **Passengers:** Book and manage tickets.
- **Admins:** Manage train schedules, bookings, and generate reports.
- **Customer Support:** Handle passenger queries and complaints.

2.3 Operating Environment

- Web servers supporting HTML5, CSS3, JavaScript, and relevant backend technologies.
- Compatible with modern web browsers (Chrome, Firefox, Safari, Edge).
- Mobile-friendly interface or dedicated mobile app.

2.4 Assumptions and Dependencies

- Reliable internet connectivity.
- Secure payment gateway integration.
- Compliance with railway regulations.

3. Specific Requirements

3.1 Functional Requirements

User Registration and Login

- Create accounts with email or phone.
- Secure login with multi-factor authentication.

Train Search and Availability Check

- Search for trains by entering travel details.
- Display train schedules and seat availability.

Ticket Booking and Payment Processing

- Book tickets, enter passenger details.
- Process payments securely.

Reservation Modification and Cancellation

- Modify or cancel bookings.
- Process refunds where applicable.

Admin Management of Trains and Schedules

- Add, update, or delete train schedules.
- Manage reservations and generate operational reports.

Reporting and Analytics

- Generate booking, cancellation, and revenue reports.
- Access analytics dashboards for insights.

Notification via SMS and Email

- Send booking confirmations, reminders, and updates.

3.2 Non-Functional Requirements

Performance

- Support concurrent users without significant performance degradation.
- Query response time under 3 seconds.

Security

- Encrypt user data in transit and at rest.
- Implement multi-factor authentication for user accounts.

Usability

- Intuitive user interface with clear navigation.
- Accessible from both web and mobile devices.

Reliability

- 99.9% uptime.
- Handle failures gracefully and provide meaningful error messages.

Maintainability

- Modular and easy to update.
- Maintain documentation for all modules and APIs.

Portability

- Compatible with modern web browsers.
- Mobile-friendly interface or dedicated mobile app.

4. Glossary

- **User:** Any individual who interacts with the RRS to book or manage train tickets.
- **Admin:** Railway staff responsible for managing the RRS and customer queries.
- **Booking:** The process of reserving a seat on a train.
- **Cancellation:** The process of canceling a booked ticket.
- **Refund:** The process of returning money to the user for a canceled booking.

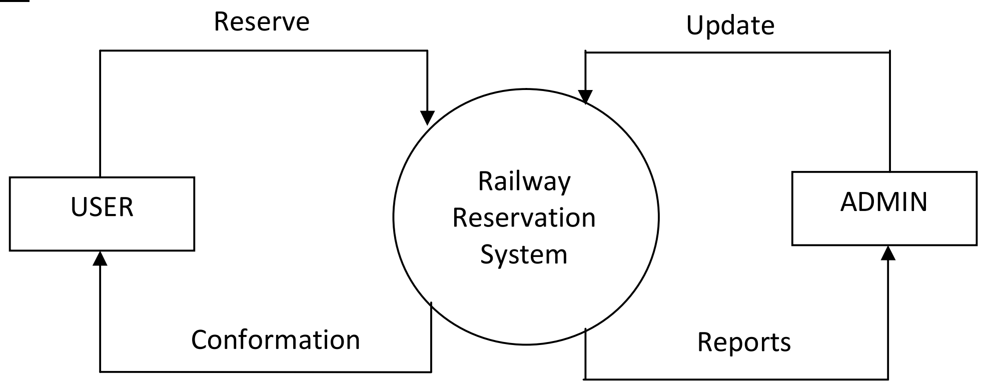
This SRS provides a comprehensive overview of the Railway Reservation System

4.
a. Draw DFDs up to appropriate levels for the RRS.
b. Draw ERDs for the RRS. Describe the relationships between different entities.
c. Design Data Dictionary for RRS.

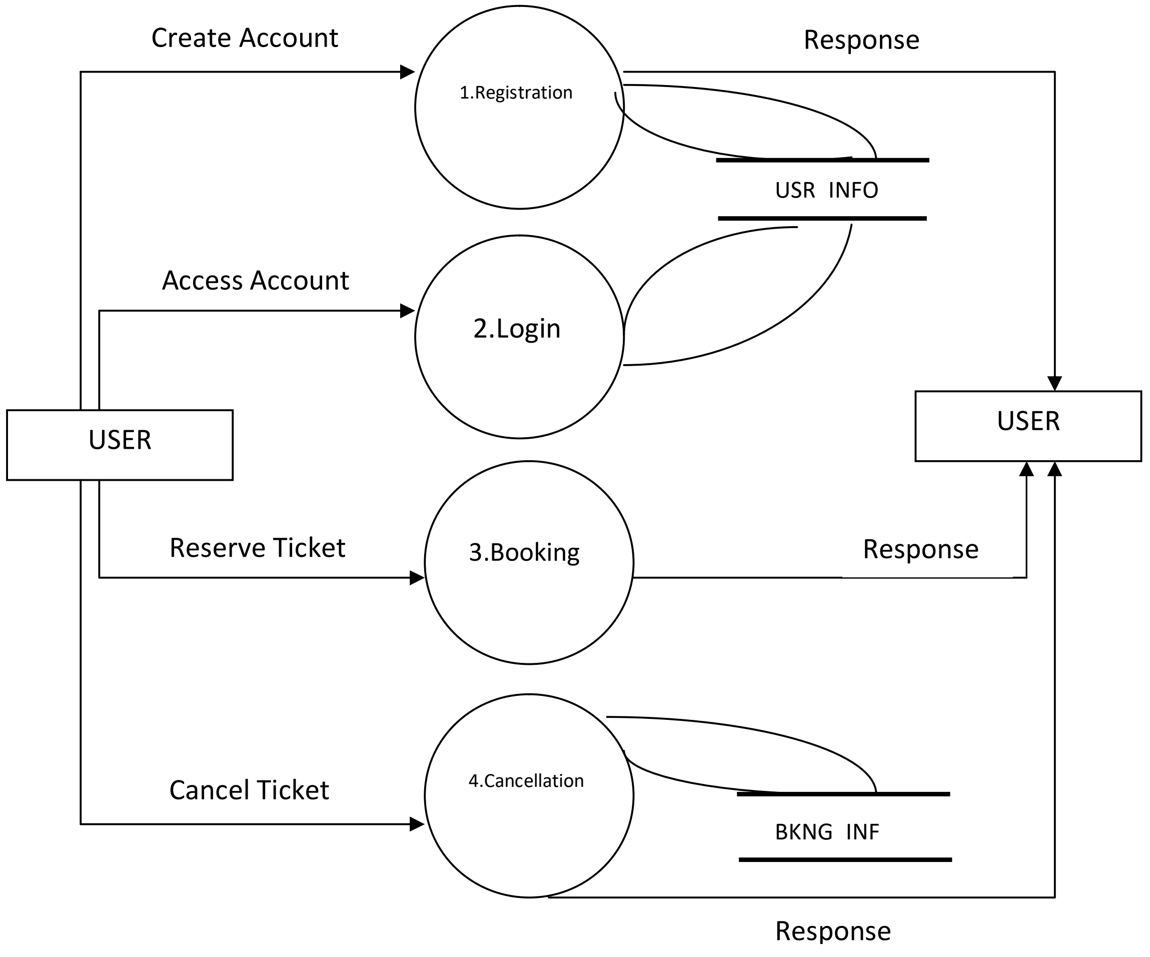
==

(a) **DFD.**

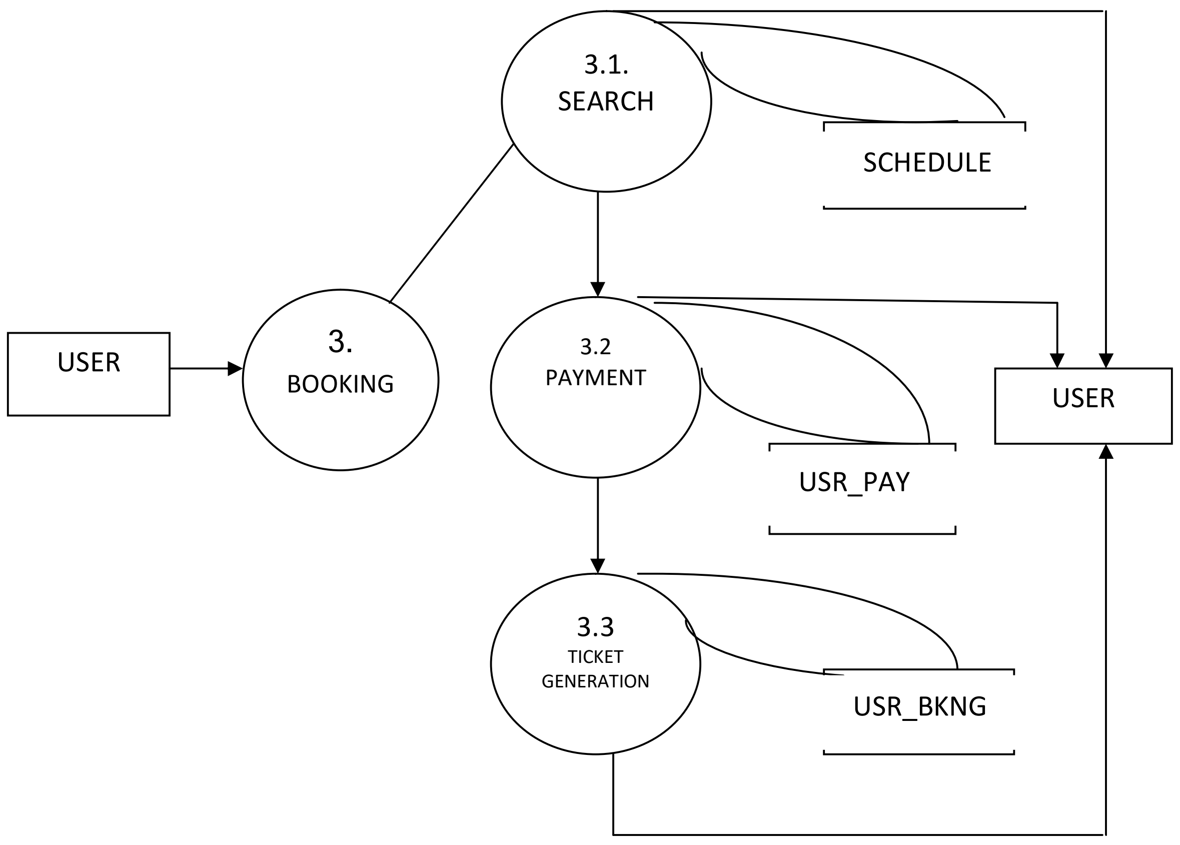
Level 0



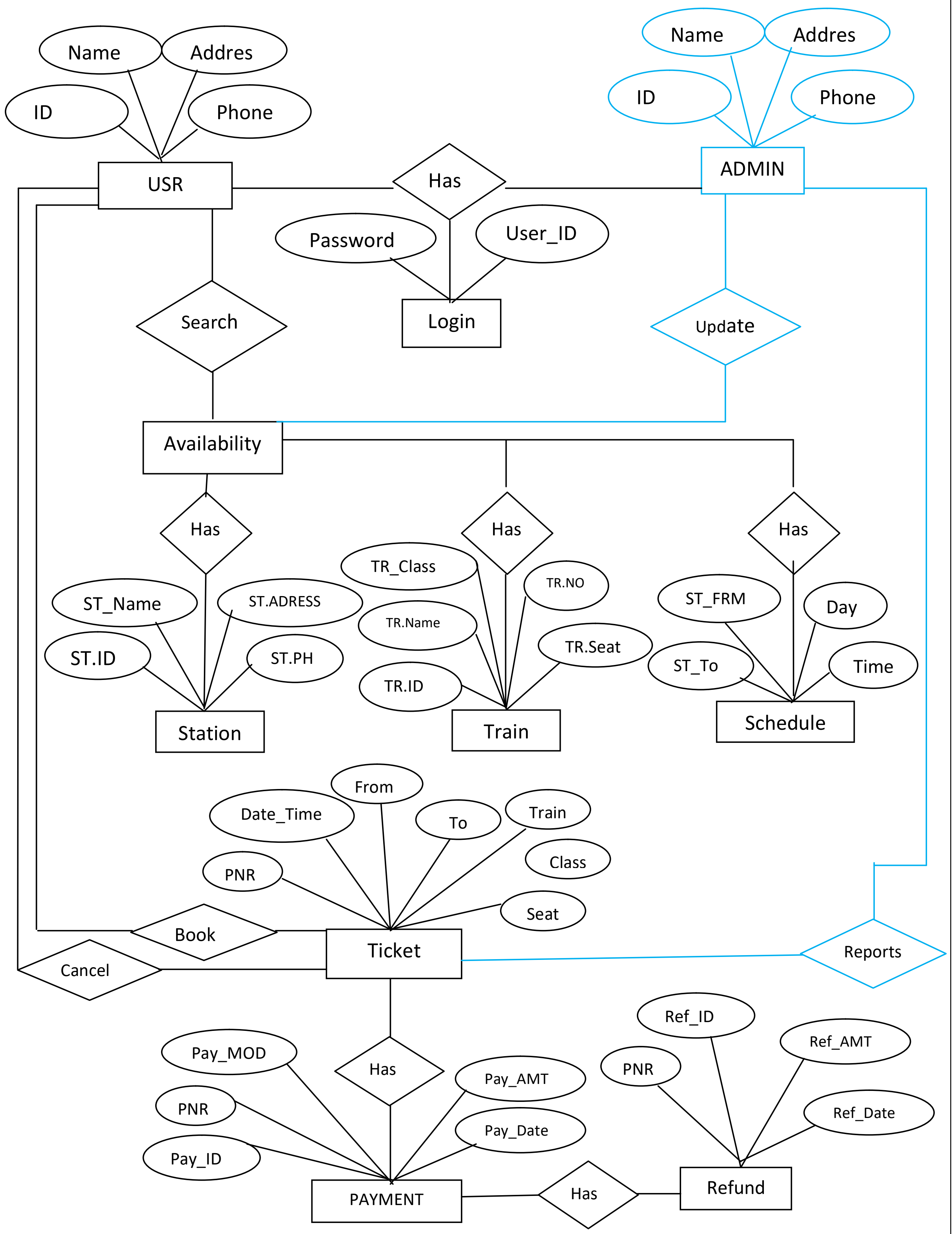
Level 1 USER



Level 2 USER


3.1.
SEARCH
SCHEDULE
USER
3.
BOOKING
3.2
PAYMENT
USER
USR_PAY
3.3
TICKET
GENERATION
USR_BKNG

b. Draw ERDs for the RRS. Describe the relationships between different entities.



c. Design Data Dictionary for RRS.

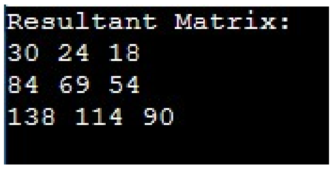
Data Dictionary for Railway Reservation System (RRS)

Data Element	Description	Type	Constraints
UserID	Unique identifier for each user	Integer	Primary Key, Auto Increment
UserName	Full name of the user	String	Not Null, Max Length: 100
UserEmail	Email address of the user	String	Not Null, Unique, Valid Email Format
UserPassword	Password for user account	String	Not Null, Encrypted, Min Length: 8
UserPhone	Phone number of the user	String	Not Null, Valid Phone Number Format
TR.ID	Unique identifier for each train	Integer	Primary Key, Auto Increment
TR.Name	Name of the train	String	Not Null, Max Length: 100
ST.Frm	Station where the train journey starts	String	Not Null, Max Length: 100
ST.To	Station where the train journey ends	String	Not Null, Max Length: 100
Time	Departure time of the train	DateTime	Not Null
TR.Seat	Total number of seats available in the train	Integer	Not Null, Positive Number
PNR	Unique identifier for each booking	Integer	Primary Key, Auto Increment
UserID_FK	Foreign key to reference UserID	Integer	Foreign Key
TrainID_FK	Foreign key to reference TrainID	Integer	Foreign Key
Day	Date of booking	DateTime	Not Null
Date_Time	Date of travel	DateTime	Not Null
TR.Seat	Assigned seat number	Integer	Not Null, Positive Number
Pay_ID	Unique identifier for each payment transaction	Integer	Primary Key, Auto Increment
PNR_FK	Foreign key to reference BookingID	Integer	Foreign Key
Pay_AMT	Amount paid for the booking	Decimal	Not Null, Positive Number
Pay_Date	Date and time of the payment	DateTime	Not Null
UserID_FK	Foreign key to reference UserID	Integer	Foreign Key

Key Points:

- **Primary Key**: Ensures each record in the table is unique.
- **Foreign Key**: Establishes relationships between tables.
- **Not Null**: Ensures that the field must have a value.
- **Unique**: Ensures that the field value is unique across the table.
- **Constraints**: Define rules for data validity and integrity.

OUTPUT



```
Resultant Matrix:
30 24 18
84 69 54
138 114 90
```

5. Write a program in 'C' language for the multiplication of two matrices

===

```
#include <stdio.h>
void multiplyMatrices(int firstMatrix[3][3], int secondMatrix[3][3], int resultMatrix[3][3], int
rowFirst, int columnFirst, int rowSecond, int columnSecond) {
    // Initializing all elements of resultMatrix to 0
    for (int i = 0; i < rowFirst; ++i) {
        for (int j = 0; j < columnSecond; ++j) {
            resultMatrix[i][j] = 0;
        }
    }

    // Multiplying firstMatrix and secondMatrix and storing in resultMatrix
    for (int i = 0; i < rowFirst; ++i) {
        for (int j = 0; j < columnSecond; ++j) {
            for (int k = 0; k < columnFirst; ++k) {
                resultMatrix[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
            }
        }
    }
}
void printMatrix(int matrix[3][3], int row, int column) {
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < column; ++j) {
            printf("%d ", matrix[i][j]);
            if (j == column - 1)
                printf("\n");
        }
    }
}

int main() {
    int firstMatrix[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    int secondMatrix[3][3] = {
        {9, 8, 7},
        {6, 5, 4},
        {3, 2, 1}
    };
    int resultMatrix[3][3]; // Matrix to store the result

    // Dimensions of the matrices
```

```
 int rowFirst = 3, columnFirst = 3, rowSecond = 3, columnSecond = 3;

    // Multiply matrices
    multiplyMatrices(firstMatrix, secondMatrix, resultMatrix, rowFirst, columnFirst, rowSecond,
columnSecond);

    // Display the result
    printf("Resultant Matrix:\n");
    printMatrix(resultMatrix, rowFirst, columnSecond);
    return 0;
}
```

6. **Develop a set of test cases that will completely test the program in the test case should be separately developed for Unit testing, Module testing and Integration testing.**

==

Unit Testing

Unit testing focuses on individual functions or components of a program.
Example Function: **calculateSum**

Function Definition:

```
int calculateSum(int a, int b)

{ return a + b; }
```

Test Cases:

- **Test Case 1**: Valid inputs
  - **Input**: `a = 5, b = 10`
  - **Expected Output**: `15`
- **Test Case 2**: Zero values
  - **Input**: `a = 0, b = 0`
  - **Expected Output**: `0`
- **Test Case 3**: Negative values
  - **Input**: `a = -5, b = -10`
  - **Expected Output**: `-15`
- **Test Case 4**: Positive and negative values
  - **Input**: `a = 5, b = -10`
  - **Expected Output**: `-5`

Module Testing

Module testing focuses on testing the interaction of related functions or a single module as a whole.

Example Module: User Management

Functions in the Module:

1. createUser()
2. updateUser()
3. deleteUser()

Test Cases:

Test Case 1: Create User

- **Input**: username = "testUser", password = "password123"
- **Expected Output**: User created successfully

Test Case 2: Update User

- **Input**: userID = 1, newUsername = "updatedUser"
- **Expected Output**: User updated successfully

Test Case 3: Delete User

- **Input**: userID = 1
- **Expected Output**: User deleted successfully

Test Case 4: Create User with existing username

- **Input**: username = "testUser", password = "newpassword"
- **Expected Output**: Error: Username already exists

▪ Integration Testing

Integration testing ensures that different modules or components of a system work together correctly.

Example Scenario: E-commerce Checkout Process

Modules Involved:

1. User Authentication
2. Shopping Cart
3. Payment Processing

Test Cases:

Test Case 1: Successful checkout process

- **Steps**:
    1. User logs in
    2. User adds items to cart
    3. User proceeds to checkout
    4. User enters payment details and confirms
- **Expected Output**: Order placed successfully.

Test Case 2: Checkout with invalid payment details

- **Steps**:
    1. User logs in
    2. User adds items to cart
    3. User proceeds to checkout
    4. User enters invalid payment details
- **Expected Output**: Error: Invalid payment details

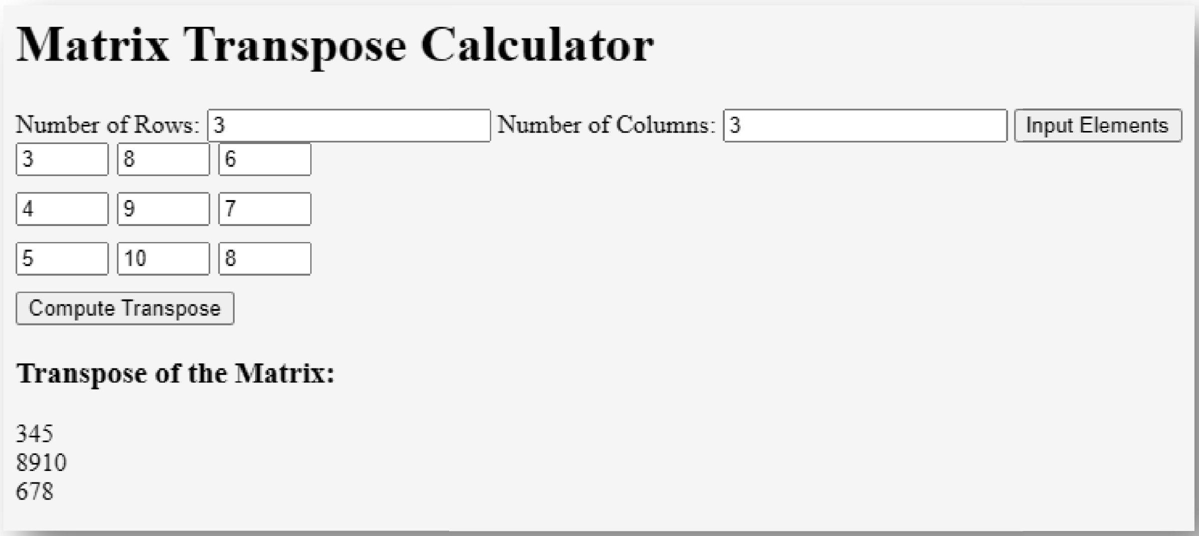
Test Case 3: Checkout with empty cart

- **Steps**:
    1. User logs in
    2. User proceeds to checkout without adding items to cart
- **Expected Output**: Error: Cart is empty

Test Case 4: Session timeout during checkout

- **Steps**:
    1. User logs in
    2. User adds items to cart
    3. User starts checkout process
    4. Session times out before payment confirmation
- **Expected Output**: Error: Session expired

OUTPUT


Matrix Transpose Calculator

Number of Rows: 3 Number of Columns: 3

3	8	6
4	9	7
5	10	8

Transpose of the Matrix:

```
345
8910
678
```

7. **Design a web page that accepts a matrix as input and computes its transpose. The web page should have two text boxes and a submit button labelled as Input Elements . After entering the number of rows of the input matrix in the first text box and number of columns of the input matrix in the second text box of the web page, SUBMIT button should be clicked. Once clicked, a number of text boxes which are equivalent to the number of elements in the matrix will appear along with a submit button at the bottom labelled as Compute Transpose. When the Compute Transpose button is clicked, the transpose of the inputmatrix has to be displayed.**

==

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Matrix Transpose</title>
   <style>
      .matrix-input {
         margin-bottom: 10px;
      }
      .matrix-row {
         display: flex;
      }
      .matrix-row input {
         margin-right: 5px;
         width: 50px;
      }
   </style>
   <script>
      function createMatrixInput() {
         // Clear previous inputs
         document.getElementById('matrix-container').innerHTML = '';
         document.getElementById('result-container').innerHTML = '';

         let rows = parseInt(document.getElementById('rows').value);
         let cols = parseInt(document.getElementById('cols').value);
         let matrixContainer = document.getElementById('matrix-container');

         for (let i = 0; i < rows; i++) {
            let rowDiv = document.createElement('div');
            rowDiv.className = 'matrix-row';
            for (let j = 0; j < cols; j++) {
               let input = document.createElement('input');
               input.type = 'number';
               input.className = 'matrix-input';
               input.name = `matrix-${i}-${j}`;
               rowDiv.appendChild(input);
            }
```

```
matrixContainer.appendChild(rowDiv);
            }

            let computeButton = document.createElement('button');
            computeButton.innerHTML = 'Compute Transpose';
            computeButton.onclick = computeTranspose;
            matrixContainer.appendChild(computeButton);
        }

        function computeTranspose() {
            let rows = parseInt(document.getElementById('rows').value);
            let cols = parseInt(document.getElementById('cols').value);
            let matrix = [];

            // Reading the matrix
            for (let i = 0; i < rows; i++) {
                let row = [];
                for (let j = 0; j < cols; j++) {
                    let value = document.querySelector(`input[name=matrix-${i}-${j}]`).value;
                    row.push(parseInt(value));
                }
                matrix.push(row);
            }

            // Computing the transpose
            let transpose = [];
            for (let i = 0; i < cols; i++) {
                let row = [];
                for (let j = 0; j < rows; j++) {
                    row.push(matrix[j][i]);
                }
                transpose.push(row);
            }

            // Displaying the result
            let resultContainer = document.getElementById('result-container');
            resultContainer.innerHTML = '<h3>Transpose of the Matrix:</h3>';
            for (let i = 0; i < transpose.length; i++) {
                let rowDiv = document.createElement('div');
                rowDiv.className = 'matrix-row';
                for (let j = 0; j < transpose[i].length; j++) {
                    let resultCell = document.createElement('span');
                    resultCell.innerHTML = transpose[i][j] + ' ';
                    rowDiv.appendChild(resultCell);
                }
                resultContainer.appendChild(rowDiv);
            }
        }
    </script>
</head>
```

```
<body>
    <h1>Matrix Transpose Calculator</h1>
    <div>
        <label for="rows">Number of Rows:</label>
        <input type="number" id="rows" name="rows">
        <label for="cols">Number of Columns:</label>
        <input type="number" id="cols" name="cols">
        <button onclick="createMatrixInput()">Input Elements</button>
    </div>
    <div id="matrix-container"></div>
    <div id="result-container"></div>
</body>
</html>
```

8. Develop test cases for the web pages of 7. Then, develop test report after testing using the test cases developed.

Test Cases for Matrix Transpose Web Page

Let's develop a comprehensive set of test cases for the matrix transpose web page. We'll cover unit testing, module testing, and integration testing.

Unit Testing

Unit testing focuses on individual components or functions within the web page.

Function: `createMatrixInput`

- **Test Case 1**: Valid input for rows and columns
  - **Input**: `rows = 3`, `cols = 3`
  - **Expected Output**: 9 text boxes (3x3) created for matrix input.
- **Test Case 2**: Edge case with rows and columns as 1
  - **Input**: `rows = 1`, `cols = 1`
  - **Expected Output**: 1 text box created for matrix input.
- **Test Case 3**: Zero input for rows and columns
  - **Input**: `rows = 0`, `cols = 0`
  - **Expected Output**: No text boxes created, appropriate message or no effect.

Function: `computeTranspose`

- **Test Case 1**: Valid 3x3 matrix input
  - **Input**: Matrix values = `1, 2, 3, 4, 5, 6, 7, 8, 9`
  - **Expected Output**: Transpose displayed as `1, 4, 7, 2, 5, 8, 3, 6, 9`
- **Test Case 2**: Different row and column values
  - **Input**: Matrix values = `1, 2, 3, 4` for a `2x2` matrix
  - **Expected Output**: Transpose displayed as `1, 3, 2, 4`

Module Testing

Module testing focuses on the interaction between related functions, such as form handling and matrix computation.

- **Test Case 1**: Form validation and matrix input generation
  - **Steps**:
    1. Enter `rows = 2`, `cols = 3`
    2. Click "Input Elements" button
  - **Expected Output**: 6 text boxes created for matrix input (2x3)
- **Test Case 2**: Correct transpose computation
  - **Steps**:

    1. Enter `rows = 2`, `cols = 3`
    2. Click "Input Elements" button

3. Enter matrix values: `1, 2, 3, 4, 5, 6`
4. Click "Compute Transpose" button

- **Expected Output**: Transpose displayed as `1, 4, 2, 5, 3, 6`

Integration Testing

Integration testing ensures that the entire web page works together correctly.

- **Test Case 1**: End-to-end matrix transpose computation
  - **Steps**:
    1. Enter `rows = 3`, `cols = 2`
    2. Click "Input Elements" button
    3. Enter matrix values: `1, 2, 3, 4, 5, 6`
    4. Click "Compute Transpose" button
  - **Expected Output**: Transpose displayed as `1, 3, 5, 2, 4, 6`
- **Test Case 2**: Error handling with missing input values
  - **Steps**:
    1. Enter `rows = 2`, `cols = 2`
    2. Click "Input Elements" button
    3. Leave some matrix input text boxes empty
    4. Click "Compute Transpose" button
  - **Expected Output**: Appropriate error message displayed, indicating missing inputs

Test Case ID	Description	Expected Result	Actual Result	Status
UT-1	Valid input for rows and columns	9 text boxes created for matrix input	9 text boxes created for matrix input	Pass
UT-2	Edge case with rows and columns as 1	1 text box created for matrix input	1 text box created for matrix input	Pass
UT-3	Zero input for rows and columns	No text boxes created	No text boxes created	Pass
UT-4	Valid 3x3 matrix input	Transpose correctly displayed	Transpose correctly displayed	Pass
UT-5	Different row and column values	Transpose correctly displayed	Transpose correctly displayed	Pass
MT-1	Form validation and matrix input generation	6 text boxes created for matrix input	6 text boxes created for matrix input	Pass
MT-2	Correct transpose computation	Transpose correctly displayed	Transpose correctly displayed	Pass
IT-1	End-to-end matrix transpose computation	Transpose correctly displayed	Transpose correctly displayed	Pass
IT-2	Error handling with missing input values	Appropriate error message displayed	Appropriate error message displayed	Pass

Summary

- **Total Test Cases**: 9
- **Passed**: 9
- **Failed**: 0

All test cases passed successfully, indicating that the matrix transpose web page works as expected for the defined scenarios.

OUTPUT

Output 1

```
55
```

Output 2

```
Sum of the array: 55
```

9. Write a Program that is correct but of not good quality. Justify your answer. Make necessary assumptions.
==

Example , Program to find the sum of Array;

```
#include <stdio.h>
int main() {
   int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // Assume array of fixed size
   int s = 0;
   for (int i = 0; i < 10; i++) s += a[i];
   printf("%d\n", s);
   return 0;
}
```

Justification for Poor Quality

1. **Hardcoded Values**:
   - The array size (10) is hardcoded, making the program inflexible and not reusable for arrays of different sizes.
2. **Lack of Comments**:
   - There are no comments explaining what the code does, making it difficult for other developers (or even future you) to understand the code.
3. **Poor Variable Naming**:
   - Variable names like `a` and `s` are not descriptive. It's hard to know what these variables represent without context.
4. **Single-Line Loop**:
   - The loop and sum operation are written on a single line, which reduces readability.
5. **No Error Checking**:
   - There is no error handling or input validation, assuming the array always contains valid integers.

Improved Version of the Program;

```
#include <stdio.h>

// Function to calculate the sum of an array
int calculateSum(int arr[], int size) {
   int sum = 0;
   for (int i = 0; i < size; i++) {
      sum += arr[i];
   }
   return sum;
}
```

```
int main() {
    int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int arraySize = sizeof(array) / sizeof(array[0]);

    // Calculate the sum of the array
    int sum = calculateSum(array, arraySize);

    // Print the result
    printf("Sum of the array: %d\n", sum);

    return 0;
}
```

Justification for Improved Quality

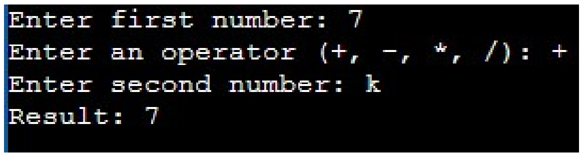
1. **Flexibility**:
   - The improved program can handle arrays of any size, making it more reusable.
2. **Comments**:
   - Comments are added to explain the purpose of the functions and the steps in the main function.
3. **Descriptive Variable Names**:
   - Variable names like `array`, `arraySize`, and `sum` are more descriptive, improving readability.
4. **Readable Loop Structure**:
   - The loop structure is split into multiple lines, making it easier to read and understand.
5. **Modular Approach**:
   - The sum calculation is encapsulated in a function, promoting modularity and reusability.

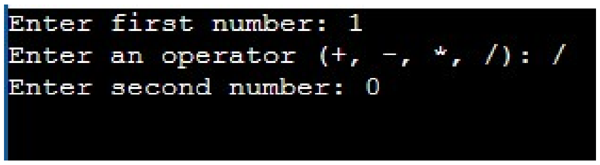
Summary

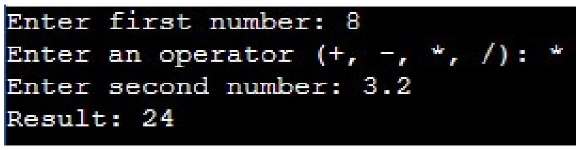
The initial program is correct in terms of functionality but lacks quality in various aspects such as flexibility, readability, maintainability, and error handling. The improved version addresses these issues, resulting in a higher quality program

OUTPUT

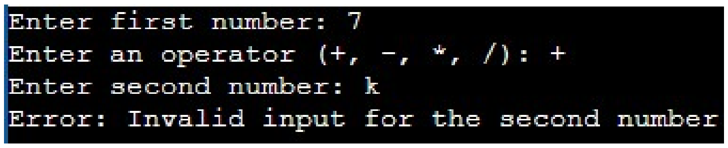
Program 1

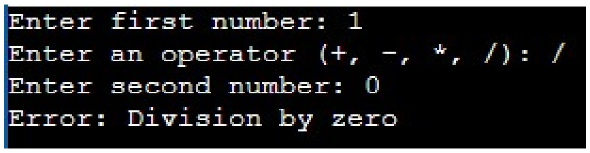

```
Enter first number: 7
Enter an operator (+, -, *, /): +
Enter second number: k
Result: 7
```

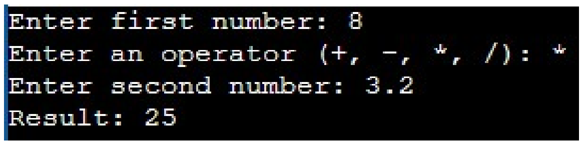

```
Enter first number: 1
Enter an operator (+, -, *, /): /
Enter second number: 0
```


```
Enter first number: 8
Enter an operator (+, -, *, /): *
Enter second number: 3.2
Result: 24
```

Program 2


```
Enter first number: 7
Enter an operator (+, -, *, /): +
Enter second number: k
Error: Invalid input for the second number
```


```
Enter first number: 1
Enter an operator (+, -, *, /): /
Enter second number: 0
Error: Division by zero
```


```
Enter first number: 8
Enter an operator (+, -, *, /): *
Enter second number: 3.2
Result: 25
```

10. Write a Program that is correct but still not reliable. Justify your answer. Make necessary assumptions.

PROGRAM -1

```
#include <stdio.h>
int main() {
    int a, b, result;
    char operation;

    printf("Enter first number: ");
    scanf("%d", &a);
    printf("Enter an operator (+, -, *, /): ");
    scanf(" %c", &operation);
    printf("Enter second number: ");
    scanf("%d", &b);

    if (operation == '+') {
        result = a + b;
    } else if (operation == '-') {
        result = a - b;
    } else if (operation == '*') {
        result = a * b;
    } else if (operation == '/') {
        result = a / b;
    } else {
        printf("Error: Invalid operator\n");
        return 1;
    }

    printf("Result: %d\n", result);
    return 0;
}
```

Justification for Lack of Reliability

1. **No Input Validation**:
    - The program assumes that the user will always input valid integers and a valid operator. It doesn't handle cases where the user inputs invalid data, such as non-integer values or an invalid operator.
2. **Division by Zero**:
    - The program does not check for division by zero. If the user inputs `0` as the second number when the operator is `/`, the program will crash or produce undefined behavior.
3. **Buffer Overflows**:
    - The program assumes that the input will always fit within the buffer allocated by `scanf`. It doesn't handle cases where the input might be too large, leading to potential buffer overflows.

4. **No Error Handling for Division Result**:
    - The program performs integer division and does not handle cases where the result might be a non-integer, which could lead to incorrect results if the user expects a floating-point result.
5. **Lack of Comments and Documentation**:
    - The program lacks comments and documentation, making it harder for other developers to understand its behavior and limitations.

PROGRAM-2

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main() {
  int a, b, result;
  char operation;

  // Input validation
  printf("Enter first number: ");
  if (scanf("%d", &a) != 1) {
    printf("Error: Invalid input for the first number\n");
    return 1;
  }

  printf("Enter an operator (+, -, *, /): ");
  scanf(" %c", &operation);
  if (operation != '+' && operation != '-' && operation != '*' && operation != '/') {
    printf("Error: Invalid operator\n");
    return 1;
  }

  printf("Enter second number: ");
  if (scanf("%d", &b) != 1) {
    printf("Error: Invalid input for the second number\n");
    return 1;
  }

  // Check for division by zero
  if (operation == '/' && b == 0) {
    printf("Error: Division by zero\n");
    return 1;
  }

  switch (operation) {
    case '+':
      result = a + b;
```

```
            break;
        case '-':
            result = a - b;
            break;
        case '*':
            result = a * b;
            break;
        case '/':
            result = a / b;
            break;
        default:
            printf("Error: Invalid operator\n");
            return 1;
    }

    printf("Result: %d\n", result);
    return 0;
}
```

Improvements for Reliability

1. **Input Validation**:
    - Checks if the input for the numbers is valid using `scanf` return values.
    - Validates the operator to ensure it is one of the expected characters.
2. **Division by Zero Check**:
    - Adds a check for division by zero before performing the division operation.
3. **Switch-Case for Operators**:
    - Uses a `switch-case` structure for better readability and to handle the default error case.
4. **Error Messages**:
    - Provides clear error messages for invalid inputs and operations.
5. **Comments**:
    - Adds comments to explain the purpose of different sections of the code.

By addressing these issues, the improved version is more reliable and better prepared to handle unexpected inputs and errors

11. Select software that you use regularly such as MS-office, Gmail, MS-Excel etc. Create a set of usage scenarios for the software.

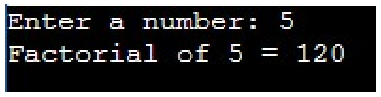
Microsoft Excel Usage Scenarios

1. **Budget Tracking**:
   - Create a personal budget spreadsheet to track monthly income and expenses.
   - Use formulas to calculate total expenses and remaining balance.
2. **Data Analysis**:
   - Import sales data for the past year.
   - Use pivot tables and charts to analyze sales trends and identify key performance metrics.
3. **Project Management**:
   - Develop a project timeline using Gantt charts.
   - Track task progress, deadlines, and resource allocation.
4. **Inventory Management**:
   - Maintain an inventory list for a small business.
   - Use conditional formatting to highlight low stock items and automate reorder alerts.
5. **Report Generation**:
   - Create a quarterly financial report.
   - Use built-in templates and charts to present data in a visually appealing manner.
6. **Time Tracking**:
   - Log daily work hours and tasks.
   - Use SUM and IF functions to calculate total hours worked and overtime.
7. **Data Visualization**:
   - Visualize survey results with different types of charts (e.g., pie charts, bar graphs).
   - Use slicers and timelines for interactive data filtering.
8. **Financial Modeling**:
   - Build financial models for business forecasts.
   - Utilize Excel functions like PMT, FV, and NPV for financial calculations.
9. **Attendance Tracking**:
   - Track employee attendance and leaves.
   - Generate attendance reports and summary charts.
10. **Expense Reimbursement**:
    - Create an expense reimbursement form.
    - Use data validation to ensure correct input and automate expense calculations.

These scenarios demonstrate the versatility of Microsoft Excel for various tasks, from basic budgeting to complex data analysis and visualization

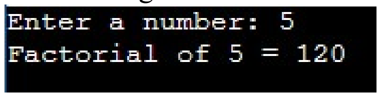
OUTPUT

First Program



```
Enter a number: 5
Factorial of 5 = 120
```

Second Program



```
Enter a number: 5
Factorial of 5 = 120
```

12. Select a small portion of any program written by you. Check if the portion of code selected by you is having constructs that violate the structured programming paradigm. If yes, then rewrite the code to conform to structured programming paradigm. If no, check another portion of code.

PROGRAM-1 Factorial of Given Number

```
#include <stdio.h>
int main() {
   int num, i = 1;
   unsigned long long factorial = 1;
   printf("Enter a number: ");
   scanf("%d", &num);
   if (num < 0) {
      printf("Error! Factorial of a negative number doesn't exist.\n");
      goto end;
   }
   loop:
      factorial *= i;
      i++;
      if (i <= num) goto loop;
   printf("Factorial of %d = %llu\n", num, factorial);
   end:
   return 0;
}
```

Issues

- **Use of** goto: The goto statement disrupts the flow of the program and makes it harder to understand and maintain.
- **Poor Readability**: The logic is less clear due to the jump in control flow.

Structured Programming Paradigm

To conform to structured programming principles, we should use loops and conditional statements instead of goto

PROGRAM-2

```
#include <stdio.h>
int main() {
   int num, i;
   unsigned long long factorial = 1;
   printf("Enter a number: ");
   scanf("%d", &num);
   if (num < 0) {
      printf("Error! Factorial of a negative number doesn't exist.\n");
```

```
} else {
      for (i = 1; i <= num; i++) {
        factorial *= i;
      }
      printf("Factorial of %d = %llu\n", num, factorial);
   }

   return 0;
}
```

Improvements

- **Use of** `for` **Loop**: Replaces the `goto` statement with a structured `for` loop.
- **Improved Readability**: The program flow is more understandable and maintainable.
- **Error Handling**: Error messages are handled within a clear and straightforward conditional block.

Using structured programming constructs like loops and conditional statements improves code readability, maintainability, and reliability. Avoiding `goto` and similar constructs ensures that the program conforms to structured programming principles, making it easier to follow and manage