CC-112L

Programming Fundamentals

Laboratory 03

Introduction to Programming, Algorithms and C

Version: 1.0.0

Release Date: 24-02-2025

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- Understand and implement different **control structures** in C.
- Write C programs using **decision-making** and **looping constructs**.

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course Instructor	Hafiz Anzar Ahmad	anzar@pucit.edu.pk
Teacher Assistants	Manahil	Bitf21m002@pucit.edu.pk
	Maheen Fatima	Bitf22m031@pucit.edu.pk
	Rimsha Majeed	Bitf22m029@pucit.edu.pk
	Momna Muzaffar	Bcsf22m021@pucit.edu.pk
	Zainab Mehmood	Bcsf22m038@pucit.edu.pk
	Khadija tul Kubra	Bitf22m025@pucit.edu.pk
	Inam ul Haq	Bitf22m017@pucit.edu.pk
	M. Saad	Bitf23m003@pucit.edu.pk

Overview

Loops are fundamental control structures in programming that allow repetitive execution of code. Among these, the **`while`** and **`do-while`** loops are particularly useful for situations where the number of iterations is unknown beforehand. Understanding their efficient use helps programmers write clear, optimized, and effective code.

---

1. while Loop:

The `while` loop executes a block of code **as long as the given condition remains true**. If the condition is false initially, the loop body will **not execute at all**.

Syntax:

```
while (condition) {
    // Code inside loop
}
```

Example: Printing numbers from 1 to 5

```c
#include <stdio.h>
int main() {
    int i = 1;
    while (i <= 5) {
        printf("%d ", i);
        i++;
    }
    return 0;
}
```

Output:

1 2 3 4 5

Use Cases:

- When the number of iterations is **unknown**.
- Checking for user input until a valid value is received.
- Processing data streams until an end condition is met.

Example 1: When the number of iterations is unknown

```c
#include <stdio.h>
int main() {
    int num = 1;
    while (num * num < 50) {
        printf("%d ", num);
        num++;
    }
    return 0;
}
```

Output:
1 4 9 16 25 36 49

Example 2: Checking for user input until a valid value is received

```
#include <stdio.h>
int main() {
    int num;
    printf("Enter a positive number: ");
    scanf("%d", &num);
    while (num <= 0) {
        printf("Invalid input. Enter again: ");
        scanf("%d", &num);
    }
    printf("Valid input received: %d\n", num);
    return 0;
}
```

Output:
1 4 9 16 25 36 49

Example 3: Processing data streams until an end condition is met

```
#include <stdio.h>
int main() {
    int data;
    printf("Enter numbers (enter -1 to stop):\n");
    scanf("%d", &data);
    while (data != -1) {
        printf("You entered: %d\n", data);
        scanf("%d", &data);
    }
    printf("End of input.\n");
    return 0;
}
```

Output:
Enter numbers (enter -1 to stop):
5
You entered: 5
6
You entered: 6
-1
End of input.

2. do-while Loop

The `do-while` loop **executes at least once**, regardless of the condition, because the condition is checked **after** the first iteration.

Syntax:

```
do {
    // Code inside loop
} while (condition);
```

Example: Getting user input until a positive number is entered

```c
#include <stdio.h>
int main() {
    int num;
    do {
        printf("Enter a positive number: ");
        scanf("%d", &num);
    } while (num <= 0);
    return 0;
}
```

Output:

Enter a positive number: -3
Enter a positive number: -8
Enter a positive number: 0
Enter a positive number: 4

Use Cases:

- Ensuring the loop body executes at least **once**.
- Menu-driven programs where user input is required before checking conditions.
- Validating input without an initial condition check.

Example 1: Ensuring the loop body executes at least once

```c
#include <stdio.h>
int main() {
    int num;
    do {
        printf("Enter a number: ");
        scanf("%d", &num);
    } while (num < 0);
    printf("You entered: %d\n", num);
    return 0;
}
```

Output:

Enter a number: -3
Enter a number: -8
Enter a number: 0
You entered : 0

Example 2: Menu-driven programs where user input is required before checking conditions

```c
#include <stdio.h>
int main() {
    int choice;
    do {
        printf("\nMenu:\n");
        printf("1. Option 1\n");
        printf("2. Option 2\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
    } while (choice != 3);
    printf("Exiting program.\n");
    return 0;
}
```

Output:

Menu:
1. Option 1
2. Option 2
3. Exit
Enter your choice: 1

Menu:
1. Option 1
2. Option 2
3. Exit
Enter your choice: 2

Menu:
1. Option 1
2. Option 2
3. Exit
Enter your choice: 3
Exiting program.

Example 3: Validating input without an initial condition check

```
#include <stdio.h>
int main() {
    int age;
    do {
        printf("Enter your age (must be 18 or older): ");
        scanf("%d", &age);
    } while (age < 18);
    printf("You are eligible.\n");
    return 0;
}
```

Output:

Enter your age (must be 18 or older): 16
Enter your age (must be 18 or older): 17
Enter your age (must be 18 or older): 18
You are eligible.

Comparison and Efficient Use

Feature	while Loop	do-while Loop
Condition Check	Before execution	After execution
Guaranteed Execution	No	Yes, at least once
Usage	When zero iterations are possible	When at least one execution is needed

Efficiency Considerations:

- **Avoid infinite loops:** Always ensure the loop condition eventually becomes false.
- **Optimize condition checks:** Repeated calculations in the condition can slow down execution.
- **Use do-while for validation tasks:** When input must be taken at least once, `do-while` is preferred.
- **Use while for unknown iteration needs:** Ideal for reading files, processing dynamic data, or waiting for an event.

PRE-LAB TASKS

Concepts Used: Nested Loops, Conditional Statements, Do-While Loop

---

TASK 01

Objective:
The objective of this task is to simulate a basic ATM (Automated Teller Machine) system using nested loops and conditional statements. Students will apply their knowledge of loops, user input handling, and decision-making structures to create an interactive ATM experience.
You are required to develop a simple ATM system that allows users to:

1. Login with a PIN (predefined in the program).
2. Perform banking transactions such as:
   - Checking account balance
   - Withdrawing money (with balance checks)
   - Depositing money
   - Exiting the ATM
3. The system should keep running until the user chooses to exit.
4. If the user enters the wrong PIN three times, the ATM should lock the account and exit.

Constraints

- The initial balance is 10,000 (hardcoded).
- The PIN is 1234 (hardcoded).
- The user has a maximum of three attempts to enter the correct PIN.
- The user cannot withdraw more money than available balance.
- Deposits should be positive amounts only.

TASK 02

Secure Communication Algorithm Using Fibonacci Primes

Objective:

The purpose of this lab exercise is to reinforce students' understanding of loops, conditional statements, and number theory by implementing a program that generates Fibonacci numbers, identifies the prime numbers among them, and calculates their sum. This scenario is inspired by a military engineer working on a secure communication algorithm that involves Fibonacci and prime numbers.

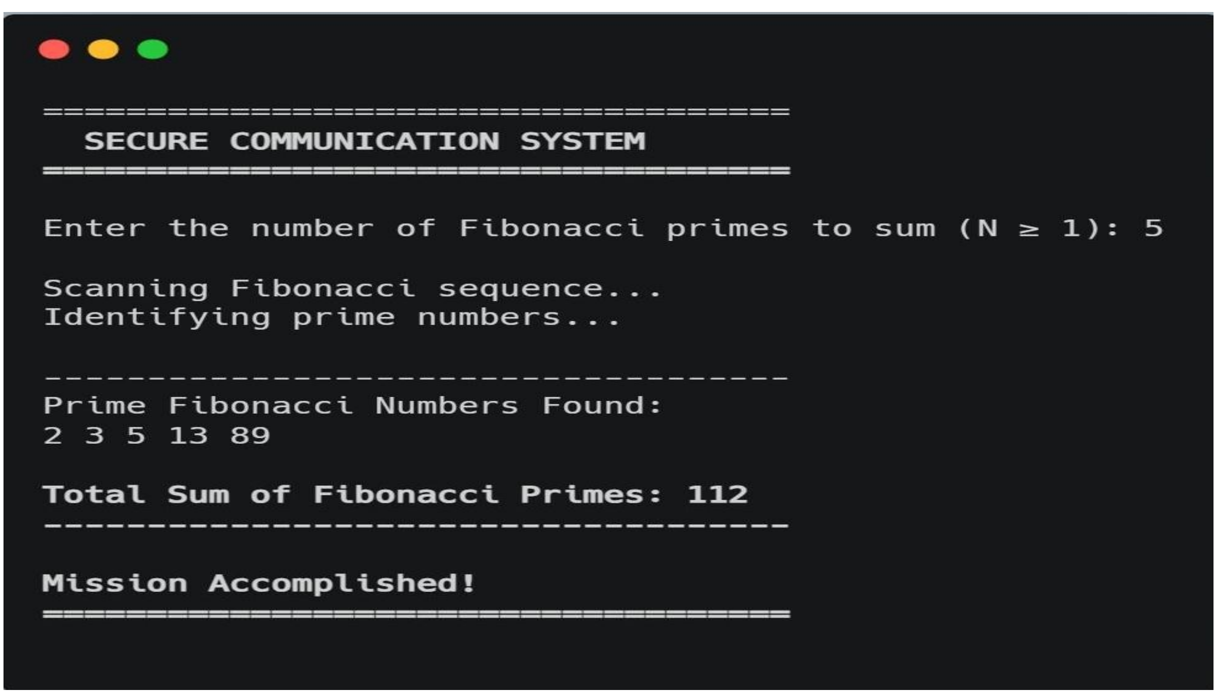
Problem Statement:

A military engineer is working on a secure communication algorithm that involves Fibonacci numbers and prime numbers. Your task is to calculate the sum of the first N Fibonacci numbers that are also prime and display them along with their sum.

$$F(n) = F(n-1) + F(n-2)$$

Start with two default numbers 0 and 1 and use the formula to calculate next numbers in the following way :-

- **F(3) = F(1) + F(2) = 0 + 1 = 1**
- **F(4) = F(2) + F(3) = 1 + 1 = 2**
- **F(5) = F(3) + F(4) = 1 + 2 = 3**



TASK 03

Minimum Steps to Transform 1 into N Objective:

The goal of this lab exercise is to develop students' problem-solving and algorithmic thinking skills by implementing an efficient method to transform the number 1 into a given non-negative integer N using the minimum number of operations. The operations allowed are:

1. **Incrementing by 1** → (N = N + 1)
2. **Doubling the number** → (N = N * 2)

Students will design an algorithm that finds the shortest possible sequence of operations to reach **N** from **1**.

Problem Statement:

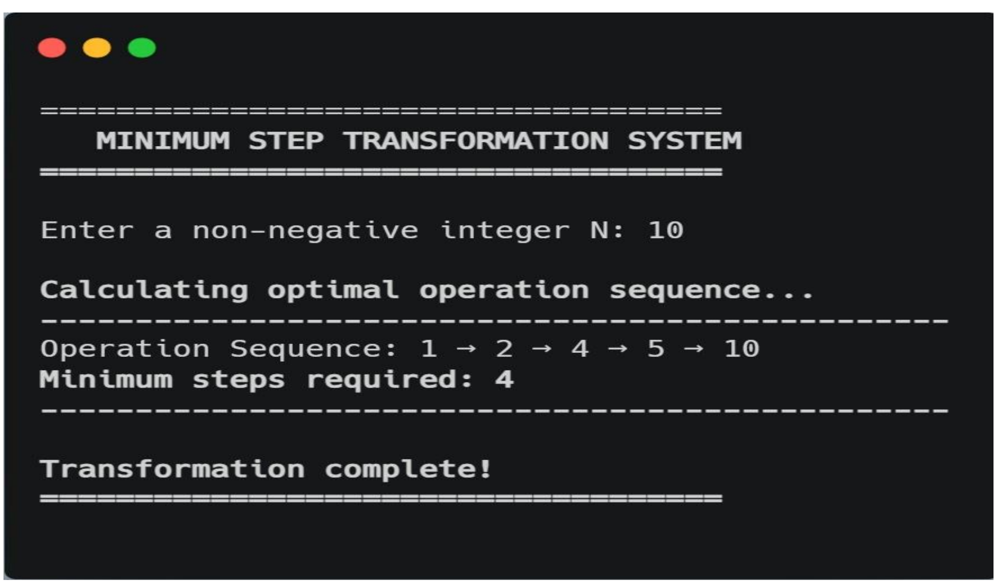
Mr. Tom is solving a programming challenge where he needs to transform the number 1 into a given non-negative integer N using the fewest possible steps. The two allowed operations are:

- **Adding 1** to the number.
- **Doubling the number.**

Given an input **N**, determine the minimum number of steps required to reach exactly **N** from **1**.

Program Execution Flow:

1. The program prompts the user to enter a non-negative integer **N**.
2. If **N = 1**, the program immediately outputs **0 steps** since no transformation is needed.
3. If **N** > **1**, the program finds the shortest sequence of operations to reach **N** from **1**.
4. The program displays the minimum number of steps taken.
5. The program may also display the exact sequence of operations performed



TASK 04

Problem Statement:

Mr. Tom is currently standing at stair 0 and wants to reach stair numbered X. He can climb either Y steps or 1 step in a single move. The task is to determine the minimum number of moves required for Mr. Tom to reach exactly stair X.

Write a program that takes **X** (destination stair number) and **Y** (maximum steps)

Execution Flow:

1. Take **X** (destination stair) and **Y** (maximum steps per move) as input.
2. Calculate the **minimum number of moves** needed using an optimal approach.
3. Print the result.

Constraints:

- The destination stair number **X** must be between **1 and 1,000,000** (inclusive).
- The maximum steps per move **Y** must be between **1 and X** (inclusive).
- The program should compute the result efficiently, ensuring minimal execution time.