

CC-112L

Programming Fundamentals

Laboratory 04

Introduction to Programming, Algorithms and C

Version: 1.0.0

Release Date: 02-03-2025

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

- Learning Objectives
- Required Resources
- General Instructions
- Overview
 - Control structures in C
 - Sequential control structures
 - Selection control structures
 - Looping control structures
 - Nested control structures
- Pre Lab tasks
 - Task 01
 - Task 02
 - Task 03

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	anzar@pucit.edu.pk
Teacher Assistants	Rimsha Majeed	bitf22m029@pucit.edu.pk
	Maheen Fatima	bitf22m031@pucit.edu.pk
	Zainab Mahmood	bcsf22m038@pucit.edu.pk
	Momina Muzaffar	bcsf22m021@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

```
#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

```
#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

```
#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

```
#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

```
#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:

Diamond Upper Half Pattern

Write a program to print the **upper half of a diamond**

number pattern:

Execution Flow:

1. Ask the user for the number of rows.
2. Use nested loops to generate the pattern.
3. Print the numbers in the given sequence with proper spacing.



```
Enter value of n: 4

 1
212
32123
4321234
```

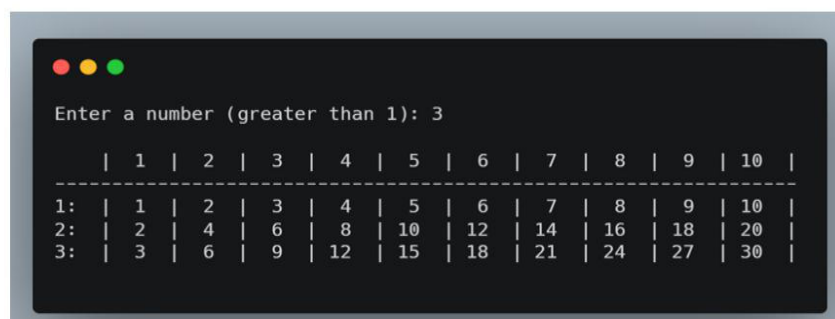
Task 02

Generate Multiplication Tables up to N

- ✓ Take input n from the user.
- ✓ Ensure $n > 1$ (valid input).
- ✓ Display multiplication tables from 1 to n for numbers 1 to n .

Execution Flow:

1. Ask the user to enter a number n .
2. Validate that $n > 1$.
3. Print multiplication tables from 1 to n .



```
Enter a number (greater than 1): 3

  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
--|---|
1: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
2: | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 |
3: | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 |
```

IN-LAB Tasks

Task 01:

(5 Marks, 10 min)

Write a C program that takes two positive integers as input and swaps their values without using any additional variables. The program should then display the swapped values.

Sample Output:

Enter 1st number: 5
Enter 2nd number: 8

After swapping:
First number = 8
Second number = 5

Task 02:

(5 Marks, 10 min)

Write a C program that takes two positive integers as input and calculates their product without using the * operator.

Sample Output:

Enter 1st number: 3
Enter 2nd number: 4
Product of 3 and 4 is 12

Task 03:

(10 Marks, 15 min)

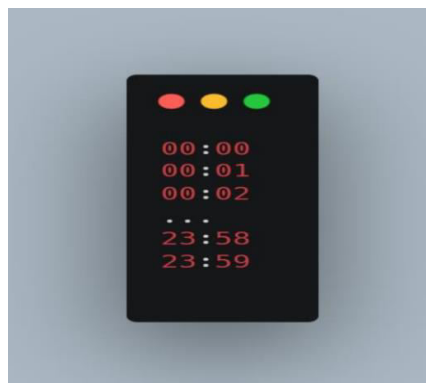
Digital Clock

Write a C program to print a 24-hour digital clock using loops.

Execution Flow: Use loops to print 00:00 to 23:59.

Constraints: No arrays or built-in time functions allowed

Sample Output:



Task 04:

(15 Marks, 20 min)

Scenario-Based Problem: Secret Lucky Discount Scheme

A store is offering a special discount scheme. Customers can join the scheme for \$5 and get a 20% discount if their bill amount matches a lucky number. However, the store does not reveal that lucky numbers are actually perfect numbers.

A perfect number is a positive integer that is equal to the sum of its proper divisors (excluding itself).

Example:

6 → Divisors: 1, 2, 3 → $1 + 2 + 3 = 6$ (Perfect Number)

10 → Divisors: 1, 2, 5 → $1 + 2 + 5 = 8$ not equal to 10 (Not a Perfect Number)

Program Flow:

1. Prompt to enter bill amount
2. Ask the user if they want to join the lucky discount scheme by paying \$5.
2. If no, then simply print the bill and terminate the program.
3. If yes, then if the sum of the digits of the bill amount is a perfect number (secret lucky number).
5. If yes, apply a 20% discount but also add the \$5 scheme fee.
6. If no, charge the full bill plus the \$5 scheme fee.

Sample Outputs:

Enter your bill amount: 50

Want to join the lucky discount scheme for \$5? (y/n): n

Your total bill is: \$50

Enter your bill amount: 28

Want to join the lucky discount scheme for \$5? (y/n): y (1+2+4+7+14 = 28 equal to bill)

Congratulations! Your bill qualifies for a lucky discount!

Your total bill is: \$27.4

(Original bill \$28 → 20% discount (\$5.6 off) → bill = 22.4 then Scheme fee \$5 added → Final bill \$27.4)

Enter your bill amount: 32

Want to join the lucky discount scheme for \$5? (y/n): y

Sorry! Your bill does not qualify for a lucky discount.

Your total bill is: \$37

POST-LAB

Switch Statement

The **switch statement** in C is an alternate to if-else-if statement which allows us to execute multiple operations for the different possible values of a single variable called switch variable. Here, we can define various statements in the multiple cases for the different values of a single variable. Switch Statement syntax:

Rules for switch statement in C language:

- The switch expression must be of an integer or character type.
- The case value must be an integer or character constant.
- The case value can be used only inside the switch statement.
- The break statement in switch case is not must. It is optional.

If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as fall through the state of C switch statement.

Functioning of switch case statement:

First, the integer expression specified in the switch statement is evaluated. This value is then matched one by one with the constant values given in the different cases. If a match is found, then all the statements specified in that case are executed along with the all the cases present after that case including the default statement. No two cases can have similar values. If the matched case contains a break statement, then all the cases present after that will be skipped, and the control comes out of the switch. Otherwise, all the cases following the matched case will be executed.

Let's see sample example:

```

1  #include <stdio.h>
2
3  int main() {
4      int studentID;
5
6      // Asking user for student ID
7      printf("Enter Student ID (1-3): ");
8      scanf("%d", &studentID);
9
10     // Switch statement to display student details
11     switch (studentID) {
12         case 1:
13             printf("Student Name: Ali\nGrade: A\n");
14             break;
15         case 2:
16             printf("Student Name: Sara\nGrade: B\n");
17             break;
18         case 3:
19             printf("Student Name: Ahmed\nGrade: C\n");
20             break;
21         default:
22             printf("Invalid Student ID! Please enter a value between 1 and 3.\n");
23     }
24
25     return 0;
26 }

```

Sample Outputs:

```

Enter Student ID (1-3): 1
Student Name: Ali
Grade: A

```

```

Enter Student ID (1-3): 4
Invalid Student ID! Please enter a value between 1 and 3.

```

Task 01

10 marks

Write a C program that displays a menu for mathematical operations:

```
PS D:\BSIT\Semester 2\F24> Select an operation:
>> 1. Addition
>> 2. Subtraction
>> 3. Multiplication
>> 4. Division
>> 5. Modulus
>> Enter your choice: |
```

Sample Output:

```
~You Select Subtraction~

Enter your choice: 2
_
```

Break and Continue Statement

Break Statement:

The **break** statement in C is used to **terminate** the loop or switch statement **immediately** when encountered. It helps in **exiting** the loop without checking further iterations.

Rules for break statement in C:

- It is generally used in loops (for, while, do-while) and switch statements.
- When encountered, the loop stops execution immediately and control moves to the next statement outside the loop.
- It helps in early termination of loops based on conditions.

Functioning of the break statement:

- When the break statement is encountered inside a loop, the loop execution is stopped immediately.
- The program resumes execution from the next statement after the loop.
- It is useful for exiting infinite loops or stopping execution when a condition is met.

Continue Statement:

The continue statement in C is used to skip the current iteration of a loop and move to the next iteration immediately. Unlike the break statement, it does not terminate the loop; it just skips the remaining code for the current iteration.

Rules for continue statement in C:

- It can only be used inside loops (for, while, do-while).
- When encountered, the remaining code inside the loop is skipped for the current iteration.
- The loop does not stop; instead, it moves to the next iteration.

Functioning of the continue statement:

- When continue is encountered, the rest of the code inside the loop is skipped for that iteration.
- The loop continues with the next iteration.
- It is useful for skipping unwanted values (e.g., ignoring negative numbers).

Here is sample Example:

```
1  #include <stdio.h>
2  int main()
3  {
4      int i;
5      for (i = 1; i <= 10; i++)
6      {
7          if (i == 5)
8          {
9              printf("Skipping number %d using continue.\n", i);
10             continue; // Skips printing 5 and moves to the next iteration
11         }
12
13         if (i == 8)
14         {
15             printf("Breaking loop at %d.\n", i);
16             break; // Exits the loop completely when i is 8
17         }
18
19         printf("Number: %d\n", i);
20     }
21
22     printf("Loop ended.\n");
23     return 0;
24 }
```


Sample Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Skipping number 5 using continue.
Number: 6
Number: 7
Breaking loop at 8.
Loop ended.
```

Task 02

10 marks

Number Filtering with Break and Continue

Write a C program that asks the user to enter numbers one by one. The program should:

- Ignore negative numbers (continue statement).
- Stop taking input if the user enters 999 (break statement).
- Display the sum of all valid numbers entered. Sample Output is here :

```
Enter numbers (999 to stop):
Enter a number: 5
Enter a number: -3
Negative number ignored.
Enter a number: 8
Enter a number: -10
Negative number ignored.
Enter a number: 12
Enter a number: 999
Sum of valid numbers: 25
PS D:\BSIT\Semester 2\F24>
```

Task 03

10 marks

Pattern Printing

Write a **C program** that prints the following **pyramid-style diamond pattern** using numbers and stars (*). The program should take an **odd integer** *n* (greater than or equal to 3) as input, which represents the number of rows in the **upper half** (excluding the middle row)

Sample output:

```
Enter an odd number (>=3) for pattern height: 5
*
 *1*
*121*
*12321*
*1234321*
*12321*
 *121*
  *1*
   *
```

Best of Luck :)