CL1002 P*rogramming Fundamentals Lab*	Lab 06 Introduction to Loops

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Fall 2024

AIMS AND OBJECTIVES

Aims:

1. **Automating Repetitive Tasks:**
   - To eliminate manual repetition by automating repetitive tasks, making the code more efficient and easier to manage.
2. **Efficient Code Execution:**
   - To enhance the efficiency of programs by reducing the amount of code required for repeating operations, leading to faster execution.
3. **Improve Code Readability and Maintainability:**
   - To improve the readability and maintainability of the code by using concise structures to handle repetitive processes rather than long, duplicated blocks of code.
4. **Control Flow Management:**
   - To give programmers more control over the flow of a program, allowing loops to repeatedly execute a block of code based on specified conditions.
5. **Dynamic Input Handling:**
   - To process dynamic inputs or continuously check conditions, allowing for flexible and adaptive behavior of programs, such as waiting for user input or processing a sequence of data elements.

Objectives:

1. **Iterate Over Data Structures:**
   - To allow for iteration over data structures like arrays, linked lists, or any other collections without writing repetitive code for each element.
2. **Terminate on Specific Conditions:**
   - To ensure that loops execute only as long as a particular condition is met, allowing the program to halt the loop once the desired outcome is achieved.
3. **Reduce Code Complexity:**
   - To simplify complex operations by reducing the number of lines of code and providing a clean and logical way to repeat tasks.
4. **Support Event-Driven Programming:**
   - To enable continuous event-checking mechanisms in real-time applications where the program needs to monitor inputs, actions, or other triggers without terminating.
5. **Flexible Program Execution:**
   - To offer flexible programming control by allowing loops to start, continue, or terminate at any point based on the evaluation of conditions (e.g., using break, continue statements).

INTRODUCTION

In programming, loops are fundamental control structures that allow a set of instructions to be repeated multiple times without writing redundant code. In C programming, loops enable developers to perform repetitive tasks efficiently, such as processing data, calculating sums, or running a program until a specific condition is met. Instead of writing the same block of code repeatedly, loops let the programmer specify a block of code to be executed multiple times based on a condition.

There are mainly two types of loops in C Programming:

Entry Controlled loops: In Entry controlled loops the test condition is checked before entering the main body of the loop. For Loop and While Loop are Entry-controlled loops.

Exit Controlled loops: In Exit controlled loops the test condition is evaluated at the end of the loop body. The loop body will execute at least once, irrespective of whether the condition is true or false. do-while Loop is Exit Controlled loop.

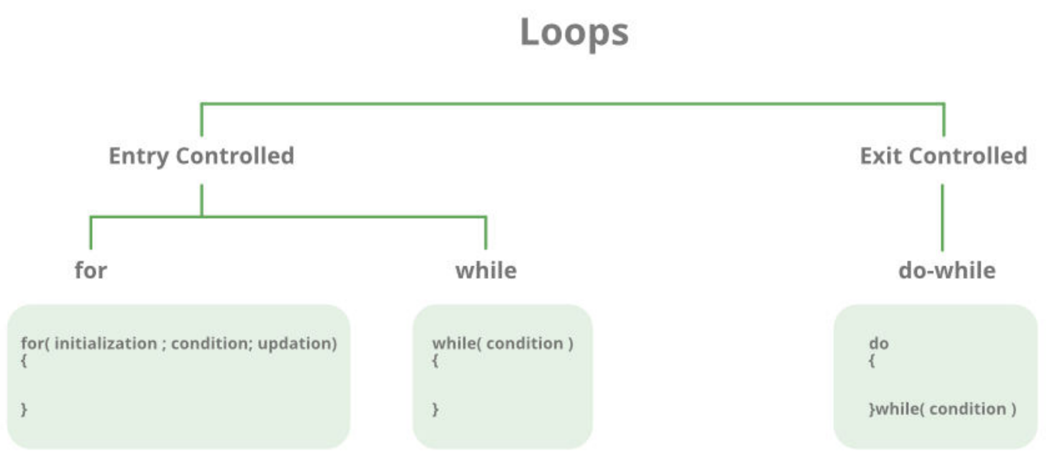


Figure 1. Types of Loops

So the types of loops that are used in C are as follows:

- **`for`**: Executes a block of code a specified number of times.
- **`while`**: Repeats as long as a condition remains true.
- **`do-while`**: Like the `while` loop, but the condition is checked after the block of code is executed, ensuring it runs at least once.

For Loop

The **for loop** in C is one of the most widely used control structures that allows a block of code to be executed repeatedly a specific number of times. It is particularly useful when the number of iterations is known beforehand. The **for** loop provides a concise way to write loops by combining initialization, condition-checking, and iteration in one line.

Syntax:

```
for (initialization; condition; increment/decrement) {
 // Block of code to be executed
}
```

Components of the for Loop:

1. **Initialization**: This is executed once at the start of the loop and is used to set the starting point of the loop. It typically involves initializing a loop control variable.
2. **Condition**: This is checked before each iteration. If the condition is true, the loop continues; if false, the loop terminates.
3. **Increment/Decrement**: This is executed after each iteration to update the loop control variable (e.g., incrementing or decrementing the counter).

Flow of Control:

1. **Initialization**: The loop control variable is initialized.
2. **Condition Checking**: The condition is evaluated. If it is true, the loop continues; otherwise, it terminates.
3. **Execution**: If the condition is true, the block of code inside the loop is executed.
4. **Increment/Decrement**: After the code block executes, the control variable is updated.
5. **Repeat**: Steps 2-4 repeat until the condition becomes false.

Flowchart

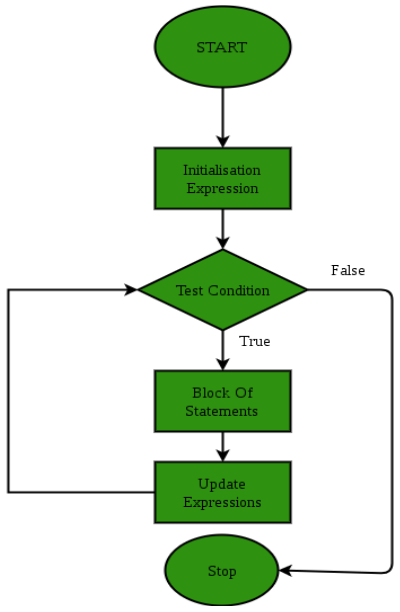


Figure 2. Equivalent Flowchart of a loop structure

EXAMPLE 1

Consider a situation where we need to print numbers 1 – 10 all in a new line

```
#include <stdio.h>

int main() {
    // Loop to print numbers from 1 to 5
    for (int i = 1; i <= 10; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

Explanation:

1. **Initialization**: int i = 1 initializes the loop control variable i to 1.
2. **Condition**: i <= 5 is checked. As long as i is less than or equal to 5, the loop continues.
3. **Increment**: After each iteration, i++ increments the value of i by 1.

WHILE LOOP

The **while loop** in C is a control flow statement that allows code to be executed repeatedly as long as a given condition is true. Unlike the for loop, where the number of iterations is usually known beforehand, the while loop is generally used when the number of iterations is not predetermined. It keeps executing the block of code until the condition evaluates to false.

Syntax:

```
while (condition) {
   // Block of code to be executed
   }
```

Components of the while Loop:

1. **Condition**: This is the expression that is evaluated before each iteration. If the condition evaluates to true, the loop continues. If it evaluates to false, the loop terminates.
2. **Loop Body**: This is the block of code that gets executed as long as the condition remains true.

Flow of Control:

1. **Condition Evaluation**: Before entering the loop, the condition is evaluated.
2. **Execution**: If the condition is true, the loop executes the block of code.
3. **Repeat**: After the loop body is executed, the condition is evaluated again. If true, the loop repeats. If false, the loop ends.

Flowchart:

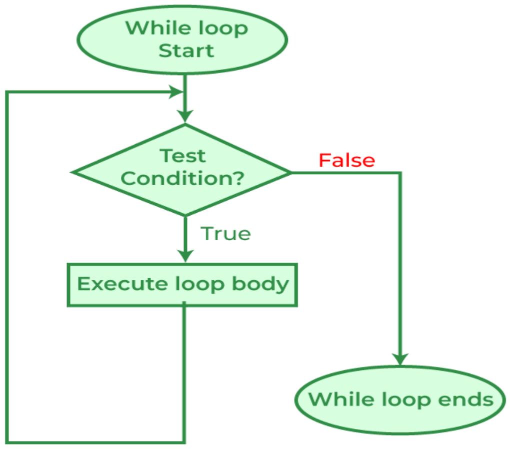


Figure 3. Equivalent Flowchart of a while loop (like for loop)

Example 2

Let's consider the same scenario from the for-loop example but with the while loop implementation.

```c
#include <stdio.h>

int main() {
    int i = 1;  // Initialization

    // Loop to print numbers from 1 to 5
    while (i <= 5) {
        printf("%d\n", i);
        i++;  // Increment
    }

    return 0;
}
```

Explanation:

1. **Initialization**: The variable i is initialized to 1.
2. **Condition**: The loop continues as long as i <= 5.
3. **Increment**: After each iteration, i++ increments the value of i by 1.

DO-WHILE LOOP

The **do-while loop** in C is similar to the while loop but with a key difference: it guarantees that the loop body is executed at least once, regardless of whether the condition is initially true or false. This is because the condition is checked **after** the loop body has executed, making it a **post-test loop**.

Syntax:

```
do {
        // Block of code to be executed
    } while (condition);
```

Components of the do-while Loop:

1. **Code Block**: The block of code that will be executed once before the condition is checked.
2. **Condition**: This is the expression that is evaluated after each execution of the code block. If it is true, the loop will continue; if false, the loop will terminate.

Flow of Control:

1. **Execution**: The loop body is executed first.
2. **Condition Evaluation**: After the execution, the condition is checked.
3. **Repeat**: If the condition is true, the loop repeats, and the code block is executed again. If false, the loop terminates.

Flowchart:

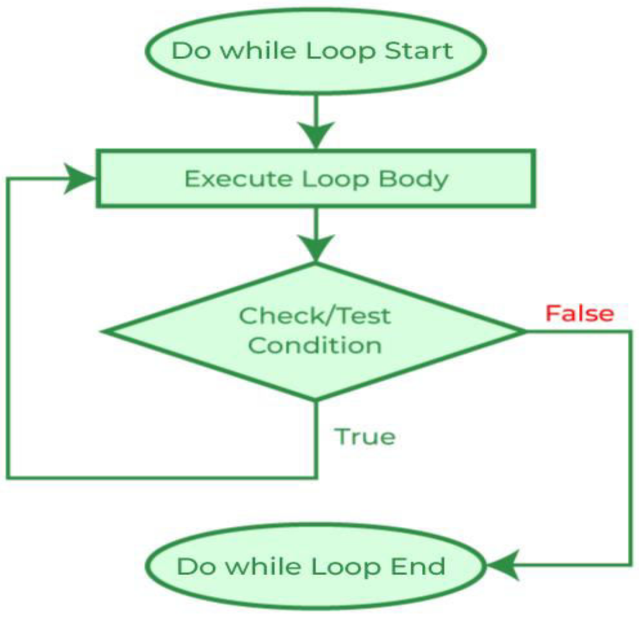


Figure 4. Equivalent Flowchart of the Do-While Loop

EXAMPLE 3

We will consider the same scenario as mentioned in the previous two examples

```c
#include <stdio.h>

   int main() {
       int i = 1;

       // Loop to print numbers from 1 to 5
       do {
           printf("%d\n", i);
           i++;  // Increment
       } while (i <= 5);

       return 0;
   }
```

Note: In Do-While loop the code will execute at least once even if the condition is false.

Explanation:

1. **Initialization**: The variable i is initialized to 1.
2. **Execution**: The code inside the do block is executed once, even if the condition is false.
3. **Condition**: After executing the block, the condition i <= 5 is checked. If it is true, the loop repeats.

Problems to Solve:

1. Which loop system would be better for user input. Justify your answer by creating a program that takes a value and adds it to a variable and prints it repeatedly until the user enters a zero value.

2. Write a program to check whether a given number is prime or not.

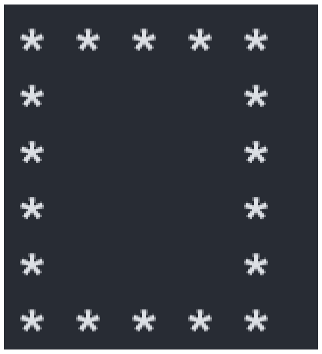
3. Using the above program integrate the number if it is a prime and print the Fibonacci series till that number.

Example
Input: 5 Output:

Number is prime
Series is = 0 1 1 2 3

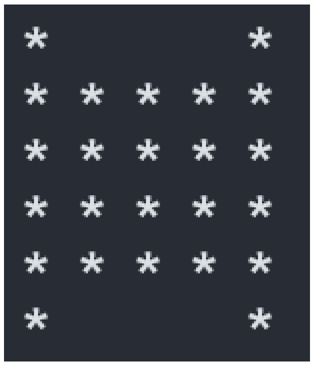
4. Write a program to check whether a number is an Armstrong number or not. An Armstrong number is a number that is equal to the sum of cubes of its digits.

5.Make a pattern mentioned below using loops



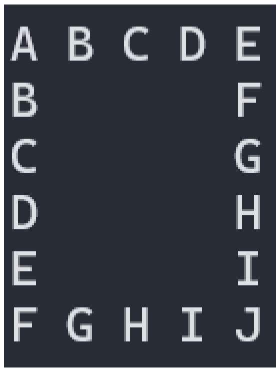
```
* * * * *
*       *
*       *
*       *
*       *
* * * * *
```

6. Make an opposite of the above-mentioned pattern using do-while loop



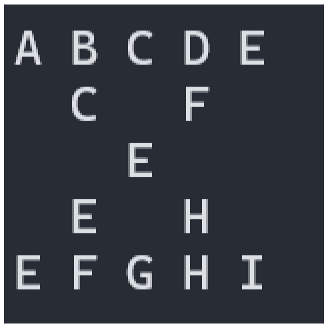
```
*       *
* * * * *
* * * * *
* * * * *
* * * * *
*       *
```

7. Make another pattern like this using characters



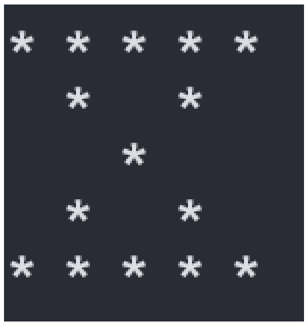
```
A B C D E
B       F
C       G
D       H
E       I
F G H I J
```

8. Make another pattern like this but in the shape of an hourglass



```
A B C D E
  C   F
    E
  E   H
E F G H I
```

Alternatively, you can make it look like this



```
* * * * *
  *   *
    *
  *   *
* * * * *
```