Experiment: 1.

Write a C program and design test cases for the following Control and decision-making statement.

a) For.. Loop b) Switch...case c) Do.. While d) If.. Else

Aim:

The C program provided serves several instructional purposes, each aimed at demonstrating the usage and functionality of different control and decision-making structures in C programming.

Objectives:

The objective for each part of the program is to write a c-program and also write a test case of each one with examples.

Let us consider,

For Loop:

- To illustrate how a for loop is used to execute a block of code a specific number of times.
- In this case, it prints numbers from 1 to 5, demonstrating how the loop iterates over a sequence of integers.

Switch Case:

- To demonstrate the switch statement, which is used for decision-making among multiple cases.
- The program includes a simple calculator that lets users choose an operation (addition, subtraction, multiplication, or division) to perform on two predefined integers.
- It showcases how the switch can handle different cases and a default case for invalid inputs.

Do..While Loop:

- To show the functionality of a do..while loop, which ensures that the loop's body is executed at least once regardless of the condition at the start.
- This part of the program outputs the starting integer value even though the continuation condition is false at the outset.
- Illustrating the loop's characteristic of checking the condition after executing the loop body.

If..Else Statement:

- To provide an example of using if..else statements for conditional execution based on the evaluation of an expression.
- The program checks whether a number is even or odd, showing basic conditional logic in action.

Each section of the program is designed to help beginner students understand and visualize how these fundamental control structures operate within a C program, making it a practical tool for learning and teaching basic programming concepts.

Below is a C program that demonstrates the use of various control and decision-making statements, specifically: for loop, switch case, do..while loop, and if..else statements. I'll also provide test cases for each statement.

a. **Program for For loop demonstration**

```
#include <stdio.h>
int main() {
        int i, choice;
        // For loop demonstration: Print numbers from 1 to 5
        printf("For loop output: ");
        for (i = 1; i <= 5; i++) {
                        printf("%d ", i);
                      }
         printf("\n");
         }
```

To design a test case for the provided C program, which demonstrates a for loop that prints numbers from 1 to 5, we'll include all necessary details such as a unique test case ID, a clear description, preconditions, test steps, expected results, and more.

Here's how you can structure this test case:

Test Case for For Loop Demonstration:

- **Test Case ID:** TC001_FL
- **Description:** Verify that the for loop correctly prints the numbers from 1 to 5 sequentially.
- **Preconditions:**
  - The program is successfully compiled without errors.
  - The executable file is ready to run.
- **Test Steps:**
  - Open a command line interface (CLI) or terminal.
  - Run the compiled executable of the program.

    - Observe the output on the console.
- **Test Data:** None required. (Because statically assigned n=5)
- **Expected Result:** The output on the console should read: "For loop output: 1 2 3 4 5 "
- **Actual Result:** This would be filled out after running the test to note what was displayed on the console.
- **Pass/Fail Criteria:** Pass if the output exactly matches the expected result, including spaces; fail otherwise.
- **Postconditions:** None.
- **Remarks:** This test will help ensure that the for loop iterates properly from 1 to 5, and that each number is printed with a space following it.

Types of Input to Consider

For this specific program, the primary concern is not the variety of inputs but ensuring that the loop executes correctly:

- **Boundary Values:** This is already hard-coded in the loop, but you'd want to ensure the loop correctly starts at 1 and ends at 5.
- **Loop Execution:** Ensure that the loop iterates the correct number of times and that the output is as expected.
- **Output Format:** Check that the output formatting (spacing, line breaks) matches the expected format.

Note on Execution:

- **Environment Setup:** Ensure that the terminal or command prompt is configured to run C programs and that the path to the compiled program is accessible.
- **Observation:** Pay close attention to the spacing and sequence of the numbers. The expected result specifies a space after each number, which must be included in the actual output for the test to pass.
- **Sensitivity to Changes:** If any modifications are made to the loop conditions or the print statement, this test case may need to be revised to reflect those changes.

This test case is straightforward but essential for confirming that basic loop functionality works as expected in a simple C program. It can be adapted for similar scenarios with variations in loop conditions or output formats.

b. Program for Switch case demonstration:

```
// Switch case demonstration: Simple calculator
int a = 10, b = 5;
printf("Enter choice (1:Add, 2:Subtract, 3:Multiply, 4:Divide): ");
scanf("%d", &choice);
switch(choice) {
                case 1:
                        printf("Addition is %d\n", a + b);
                        break;
                case 2:
                        printf("Subtraction is %d\n", a - b);
                        break;
                case 3:
                        printf("Multiplication is %d\n", a * b);
                        break;
                case 4:
                        printf("Division is %d\n", a / b);
                        break;
                default:
                        printf("Invalid choice\n");
            }
```

For the provided C program that demonstrates a simple calculator using a switch case structure, we can design a series of test cases to validate each arithmetic operation and also handle an invalid choice scenario. Each test case ensures that the calculator handles user inputs correctly and performs the expected operation.

Here are the test cases:

1. Test Case for Addition

- **Test Case ID:** TC_CALC01
- **Description:** Verify that selecting the addition operation correctly adds two predefined numbers.
- **Preconditions:** The calculator program is compiled and running.
- **Test Steps:**
  - Run the program.
  - Enter 1 for addition when prompted.
  - Test Data: choice = 1
- **Expected Result:** The output should be "Addition is 15".
- **Actual Result:** Needs to be recorded during testing.
- **Pass/Fail Criteria:** Pass if the output matches the expected result.

- **Postconditions:** None.
- **Remarks:** Tests basic addition functionality.

2. Test Case for Subtraction

- **Test Case ID:** TC_CALC02
- **Description:** Verify that selecting the subtraction operation correctly subtracts two predefined numbers.
- **Preconditions:** The calculator program is compiled and running.
- **Test Steps:**
  - Run the program.
  - Enter 2 for subtraction when prompted.
  - Test Data: choice = 2
- **Expected Result:** The output should be "Subtraction is 5".
- **Actual Result:** Needs to be recorded during testing.
- **Pass/Fail Criteria:** Pass if the output matches the expected result.
- **Postconditions:** None.
- **Remarks:** Tests basic subtraction functionality.

3. Test Case for Multiplication

- **Test Case ID:** TC_CALC03
- **Description:** Verify that selecting the multiplication operation correctly multiplies two predefined numbers.
- **Preconditions:** The calculator program is compiled and running.
- **Test Steps:**
  - Run the program.
  - Enter 3 for multiplication when prompted.
  - Test Data: choice = 3
- **Expected Result:** The output should be "Multiplication is 50".
- **Actual Result:** Needs to be recorded during testing.
- **Pass/Fail Criteria:** Pass if the output matches the expected result.
- **Postconditions:** None.
- **Remarks:** Tests basic multiplication functionality.

4. Test Case for Division

- **Test Case ID:** TC_CALC04
- **Description:** Verify that selecting the division operation correctly divides two predefined numbers.
- **Preconditions:** The calculator program is compiled and running.
- **Test Steps:**

    - Run the program.
    - Enter 4 for division when prompted.
    - Test Data: choice = 4
- **Expected Result:** The output should be "Division is 2".
- **Actual Result:** Needs to be recorded during testing.
- **Pass/Fail Criteria:** Pass if the output matches the expected result.
- **Postconditions:** None.
- **Remarks:** Tests basic division functionality.

5. Test Case for Invalid Choice

- **Test Case ID:** TC_CALC05
- **Description**: Verify that entering an invalid choice displays an appropriate error message.
- **Preconditions:** The calculator program is compiled and running.
- **Test Steps:**
    - Run the program.
    - Enter a number not listed in the choices (e.g., 5).
    - Test Data: choice = 5
- **Expected Result:** The output should be "Invalid choice".
- **Actual Result:** Needs to be recorded during testing.
- **Pass/Fail Criteria:** Pass if the output matches the expected result.
- **Postconditions:** None.
- **Remarks:** Tests error handling for invalid user input.

These test cases will help ensure that each function of the calculator works as expected and that the program handles invalid input appropriately.

This comprehensive testing approach helps in verifying that the basic functionalities of a simple calculator are robust and error-free.

c. Program for Do..while loop demonstration

```
// Do..while loop demonstration: Execute at least once
i = 10;
printf("Do..While loop output: ");
    do {
         printf("%d ", i);
         i++;
        } while (i <= 5);
printf("\n");
```

To effectively test the C program that demonstrates the behavior of a do..while loop, we can design a test case that checks the functionality of this loop type. The do..while loop guarantees that the body of the loop is executed at least once, regardless of the initial condition. Here, even though the loop's condition i <= 5 is false on the first check (since i start at 10), the loop still executes once.

Test Case for Do..While Loop Demonstration

- **Test Case ID:** TC_DW001
- **Description:** Verify that the do..while loop executes at least once despite the initial condition being false.
- **Preconditions:** The program is compiled and ready to run.
- **Test Steps:**
  - Start the program.
  - Observe the output produced by the do..while loop.
- **Test Data:** None specific beyond the initial i = 10.
- **Expected Result:** The output on the console should be "Do..While loop output: 10 ".
- **Actual Result:** This should be filled out after running the test, noting the actual output displayed on the console.
- **Pass/Fail Criteria:** Pass if the output matches the expected result, specifically displaying the number 10 followed by a space.
- **Postconditions:**
  - The program returns to the command prompt or operating system without crashing or freezing.
  - No changes in the system or environment should occur due to running this test.
- **Remarks**: This test case is critical for verifying that the do..while loop functions correctly, particularly in ensuring that the loop body is executed once before the condition is checked.

Importance of this Test Case

- **Core Functionality Check:** This test ensures that one of the fundamental aspects of the do..while loop executing at least once before condition checking is preserved, which is crucial for scenarios where at least one iteration is necessary regardless of conditions.
- **Simple Yet Effective:** Although the test is straightforward, it effectively confirms the loop's behavior, which could be foundational in more complex programming scenarios where decisions after the first iteration could affect outcomes significantly.

By focusing on the expected behavior and ensuring the environment remains stable and unchanged, this test case helps ensure the integrity and correctness of programs using do..while loops in potentially more critical applications.

d. **Program for demonstration: Check even or odd.**

```
   // If..else demonstration: Check even or odd
{
   int num = 4;
   printf("If..Else output: ");
                if (num % 2 == 0) {
                printf("%d is even.\n", num);
                } else {
                printf("%d is odd.\n", num);
                        }
   return 0;
}
```

To thoroughly test the C program designed to determine if a number is even or odd using an if..else structure, you can create separate test cases for both conditions (even and odd numbers). Here's a detailed test case design for each scenario:

Test Case for Checking Even Number

- **Test Case ID:** TC_EO001
- **Description:** Verify that the program correctly identifies an even number.
- **Preconditions:** The program is compiled and ready to run.
- **Test Steps:**
  - Run the program with num set to an even number (e.g., 4).
  - Observe the output.
  - Test Data: num = 4
- **Expected Result:** The output on the console should be "4 is even."
- **Actual Result:** This would be filled out after running the test to record the actual output displayed.
- **Pass/Fail Criteria:** Pass if the output exactly matches the expected result. Fail otherwise.
- **Postconditions:** The program terminates without errors and returns to the operating system.
- **Remarks:** This test checks the basic functionality of arithmetic operations and conditional logic in the program.

Test Case for Checking Odd Number

- **Test Case ID:** TC_EO002
- **Description:** Verify that the program correctly identifies an odd number.
- **Preconditions:** The program is compiled and ready to run.
- **Test Steps:**
    - Modify the program to set num to an odd number (e.g., 5) or recompile if necessary.
    - Run the program.
    - Observe the output.
    - Test Data: num = 5
- **Expected Result:** The output on the console should be "5 is odd."
- **Actual Result:** This would be filled out after running the test to record the actual output displayed.
- **Pass/Fail Criteria:** Pass if the output exactly matches the expected result. Fail otherwise.
- **Postconditions:** The program terminates without errors and returns to the operating system.
- **Remarks:** This test checks the program's ability to handle simple conditional branches effectively.

General Considerations for Both Test Cases

Coverage: These test cases cover both branches of the conditional logic, ensuring comprehensive testing of the program's functionality.

Simplicity: The tests are straightforward, reflecting the program's simplicity but are critical for verifying its logical correctness.

Repeatability: These tests are designed to be easily repeatable, allowing for consistent verification across multiple executions.

By testing both even and odd numbers, you ensure that the program accurately implements its intended functionality, handling both scenarios as expected. This testing approach is fundamental but effective for validating basic conditional logic in software development.

Test Case Design:

Below is an outline for designing test cases for this program:

Test Case 1: Even Number

- Input: num = 4
- Expected Output: "4 is even."
- Purpose: Verifies that the program correctly identifies an even number.

Test Case 2: Odd Number

- Input: num = 5
- Expected Output: "5 is odd."
- Purpose: Confirms that the program accurately recognizes an odd number.

Test Case 3: Zero

- Input: num = 0
- Expected Output: "0 is even."
- Purpose: Zero is technically an even number, and this case checks if the program handles this correctly.

Test Case 4: Negative Even Number

- Input: num = -2
- Expected Output: "-2 is even."
- Purpose: Ensures the program correctly evaluates negative even numbers.

Test Case 5: Negative Odd Number

- Input: num = -3
- Expected Output: "-3 is odd."
- Purpose: Tests if the program correctly evaluates negative odd numbers.

Test Case 6: Large Even Number

- Input: num = 1000000
- Expected Output: "1000000 is even."
- Purpose: Checks the program's performance with very large even numbers.

Test Case 7: Large Odd Number

- Input: num = 1000001
- Expected Output: "1000001 is odd."
- Purpose: Checks the program's performance with very large odd numbers.

Note:

- **Edge Cases:** Zero and negative numbers are typical edge cases in arithmetic-based logic. They can often reveal flaws in logical conditions.
- **Large Numbers:** Testing with large numbers ensures that there are no issues with data types or overflow.

- **Code Coverage:** These test cases ensure that both branches of the if..else statement are executed, providing full branch coverage.

These test cases can be implemented either manually, by changing the value of num and observing the output, or programmatically, by writing a test script that sets the number and checks the output against the expected result.

Evaluation:

The designed test cases ensure comprehensive coverage of each program's functionality:

Functionality Verification: Each test case confirms that specific functionalities work as expected under normal and boundary conditions.

Input Sensitivity and Output Accuracy: The test cases for the switch case and if..else statement validate the program's response to varied inputs and ensure accurate output.

Loop Execution Validation: The test cases for the loops (for and do..while) confirm correct loop execution and adherence to loop conditions.

Conclusion:

- The experiment successfully demonstrates the effectiveness of structured testing in validating the functionality of basic control structures in C programming.
- It also highlights the importance of precise and thorough test case design in ensuring program correctness and reliability.

The outcome of the experiment:

- The outcome of experiment 1 provides a comprehensive and instructive demonstration of the implementation and validation of fundamental control and decision-making structures in C programming.
- This outcome ensures that beginners can clearly understand how each control structure operates within a C program and confirms the reliability of the code through systematic testing.

Exercise assignments:

- Write a c program to generate Fibonacci numbers up to the nth series.
- Write a c program to perform the operation menu chooser option.
- Write a c program to calculate the factorial of a given number.
- Write a c program to perform prime number checking.