SRM Institute of Science and Technology

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Department of Computer Applications

Circular – 2023-24

B.Sc. CS 1ST Sem

Lab Manual: Programming for Problem Solving (USA23101J)

Lab 1: Algorithm, Flow Chart, Pseudo code

Title

Introduction to Algorithms, Flowcharts, and Pseudocode

Aim

To understand the fundamental concepts of algorithms, flowcharts, and pseudocode as tools for problem-solving in programming.

Procedure

1. **Define the Problem:** Clearly state the problem that needs to be solved.
2. **Develop an Algorithm:** Write a step-by-step procedure to solve the problem.
3. **Create a Flowchart:** Draw a graphical representation of the algorithm using standard flowchart symbols.
4. **Write Pseudocode:** Express the algorithm in a high-level, language-independent textual format.
5. **Review and Refine:** Check the algorithm, flowchart, and pseudocode for correctness and efficiency.

Source Code

No source code for this lab as it focuses on conceptual understanding and design tools.

Input

Not applicable for this lab.

Expected Output

Not applicable for this lab, as the output will be the algorithm, flowchart, and pseudocode documents.

Lab 2: Input and Output Statements

Title

Using Basic Input and Output Statements in C

Aim

To learn how to use standard input (`scanf()`) and output (`printf()`) functions in C programming to interact with the user.

Procedure

1. **Include Header:** Start by including the `stdio.h` header file.
2. **Declare Variables:** Declare variables to store the input values.
3. **Prompt for Input:** Use `printf()` to display a message prompting the user to enter data.
4. **Read Input:** Use `scanf()` to read data from the keyboard and store it in the declared variables. Ensure correct format specifiers are used.
5. **Display Output:** Use `printf()` to display the processed data or results to the console.

Source Code

```
#include <stdio.h> // Required for input/output functions

int main() {
    int age;        // Declare an integer variable for age
    char name[50];  // Declare a character array for name

    // Prompt the user to enter their name
    printf("Enter your name: ");
    // Read the name from the user. %s reads a string.
    // No & needed for character arrays when reading strings.
    scanf("%s", name);

    // Prompt the user to enter their age
    printf("Enter your age: ");
    // Read the age from the user. %d reads an integer.
    // &age provides the memory address of the age variable.
    scanf("%d", &age);

    // Display the entered name and age
    printf("Hello, %s! You are %d years old.\n", name, age);

    return 0; // Indicate successful execution
}
```

Input

```
Enter your name: Alice
Enter your age: 30
```

Expected Output

```
Hello, Alice! You are 30 years old.
```

Lab 3: Data Types

Title

Understanding Fundamental Data Types in C

Aim

To explore and understand the different fundamental data types available in C programming (e.g., `int`, `float`, `char`, `double`) and their usage.

Procedure

1. **Declare Variables:** Declare variables of different data types.
2. **Assign Values:** Assign appropriate values to these variables.
3. **Print Values and Sizes:** Use `printf()` to display the values stored in variables and their sizes using the `sizeof()` operator.
4. **Observe Output:** Note how different data types store and represent data, and their memory consumption.

Source Code

```
#include <stdio.h> // Required for input/output functions

int main() {
    // Declare and initialize variables of different data types
    int integerVar = 10;
    float floatVar = 20.5f; // 'f' suffix indicates a float literal
    double doubleVar = 30.123456789;
    char charVar = 'A';

    // Print the values and sizes of each variable
    printf("Integer Variable: %d, Size: %zu bytes\n", integerVar,
sizeof(integerVar));
    printf("Float Variable: %.2f, Size: %zu bytes\n", floatVar,
sizeof(floatVar)); // .2f for 2 decimal places
    printf("Double Variable: %.9lf, Size: %zu bytes\n", doubleVar,
sizeof(doubleVar)); // .9lf for 9 decimal places
    printf("Character Variable: %c, Size: %zu bytes\n", charVar,
sizeof(charVar));

    return 0; // Indicate successful execution
}
```

Input

No explicit input is required for this program.

Expected Output

```
Integer Variable: 10, Size: 4 bytes
Float Variable: 20.50, Size: 4 bytes
Double Variable: 30.123456789, Size: 8 bytes
Character Variable: A, Size: 1 bytes
```

(Note: The exact size in bytes might vary slightly depending on the compiler and system architecture, but the relative sizes will remain consistent.)

Lab 4: Operators and Expressions

Title

Working with Operators and Expressions in C

Aim

To understand and implement various types of operators in C (arithmetic, relational, logical, assignment, etc.) and form expressions.

Procedure

1. **Declare Variables:** Declare variables to hold operands.
2. **Perform Operations:** Apply different operators to variables to form expressions.
3. **Print Results:** Display the results of these expressions using `printf()`.
4. **Observe Precedence:** Pay attention to operator precedence and associativity.

Source Code

```
#include <stdio.h> // Required for input/output functions

int main() {
    int a = 10, b = 3;
    int sum, difference, product, quotient, remainder;

    // Arithmetic Operators
    sum = a + b;          // Addition
    difference = a - b;   // Subtraction
    product = a * b;      // Multiplication
    quotient = a / b;     // Division (integer division)
    remainder = a % b;    // Modulo (remainder)

    printf("Arithmetic Operations:\n");
    printf("a + b = %d\n", sum);
    printf("a - b = %d\n", difference);
    printf("a * b = %d\n", product);
    printf("a / b = %d\n", quotient);
    printf("a %% b = %d\n", remainder); // %% to print a literal %

    // Relational Operators (return 0 for false, 1 for true)
    printf("\nRelational Operations:\n");
    printf("a > b : %d\n", a > b);   // Greater than
    printf("a < b : %d\n", a < b);   // Less than
    printf("a == b : %d\n", a == b); // Equal to
    printf("a != b : %d\n", a != b); // Not equal to

    // Logical Operators (&&, ||, !)
    int x = 5, y = 10;
    printf("\nLogical Operations:\n");
    // (x > 0) is true, (y < 20) is true, so true && true is true (1)
    printf("(x > 0 && y < 20) : %d\n", (x > 0 && y < 20));
    // (x > 10) is false, (y < 5) is false, so false || false is false (0)
    printf("(x > 10 || y < 5) : %d\n", (x > 10 || y < 5));
    // !(x == 5) is !(true) which is false (0)
    printf("!(x == 5) : %d\n", !(x == 5));

    // Assignment Operators
    int c = 10;
    c += 5; // c = c + 5;
    printf("\nAssignment Operations:\n");
```

```
    printf("c after c += 5: %d\n", c); // c is now 15

    c *= 2; // c = c * 2;
    printf("c after c *= 2: %d\n", c); // c is now 30

    return 0; // Indicate successful execution
}
```

Input

No explicit input is required for this program.

Expected Output

```
Arithmetic Operations:
a + b = 13
a - b = 7
a * b = 30
a / b = 3
a % b = 1

Relational Operations:
a > b : 1
a < b : 0
a == b : 0
a != b : 1

Logical Operations:
(x > 0 && y < 20) : 1
(x > 10 || y < 5) : 0
!(x == 5) : 0

Assignment Operations:
c after c += 5: 15
c after c *= 2: 30
```

Lab 5: Control Statements

Title

Implementing Conditional and Loop Control Statements

Aim

To understand and implement various control flow statements in C, including `if-else`, `switch-case`, `for`, `while`, and `do-while` loops.

Procedure

1. **Conditional Statements:**
   - Use `if-else` to execute code blocks based on a condition.
   - Use `switch-case` for multi-way branching based on an integer or character expression.
2. **Looping Statements:**
   - Use `for` loop for iterating a fixed number of times.
   - Use `while` loop for iterating as long as a condition is true.
   - Use `do-while` loop for iterating at least once, then as long as a condition is true.
3. **Test Cases:** Test with different inputs to verify the behavior of each control statement.

Source Code

```
#include <stdio.h> // Required for input/output functions

int main() {
    int num;

    // --- If-Else Statement ---
    printf("Enter an integer: ");
    scanf("%d", &num);

    if (num % 2 == 0) {
        printf("%d is an even number.\n", num);
    } else {
        printf("%d is an odd number.\n", num);
    }

    // --- Switch-Case Statement ---
    char grade;
    printf("\nEnter a grade (A, B, C, D, F): ");
    // Use ' %c' to consume any leftover newline character from previous
input
    scanf(" %c", &grade);

    switch (grade) {
        case 'A':
        case 'a':
            printf("Excellent!\n");
            break;
        case 'B':
        case 'b':
            printf("Very Good!\n");
            break;
        case 'C':
        case 'c':
            printf("Good.\n");
            break;
```

```c
        case 'D':
        case 'd':
            printf("Pass.\n");
            break;
        case 'F':
        case 'f':
            printf("Fail.\n");
            break;
        default:
            printf("Invalid grade entered.\n");
    }

    // --- For Loop ---
    printf("\nNumbers from 1 to 5 using for loop:\n");
    for (int i = 1; i <= 5; i++) {
        printf("%d ", i);
    }
    printf("\n");

    // --- While Loop ---
    printf("\nNumbers from 5 to 1 using while loop:\n");
    int j = 5;
    while (j >= 1) {
        printf("%d ", j);
        j--;
    }
    printf("\n");

    // --- Do-While Loop ---
    printf("\nNumbers from 10 to 12 using do-while loop:\n");
    int k = 10;
    do {
        printf("%d ", k);
        k++;
    } while (k <= 12);
    printf("\n");

    return 0; // Indicate successful execution
}
```

Input

```
Enter an integer: 7
Enter a grade (A, B, C, D, F): B
```

Expected Output

```
7 is an odd number.

Enter a grade (A, B, C, D, F): B
Very Good!

Numbers from 1 to 5 using for loop:
1 2 3 4 5

Numbers from 5 to 1 using while loop:
5 4 3 2 1

Numbers from 10 to 12 using do-while loop:
10 11 12
```

Lab 6: Arrays- One Dimensional

Title

Working with One-Dimensional Arrays

Aim

To understand how to declare, initialize, and access elements of a one-dimensional array in C.

Procedure

1. **Declaration:** Declare a one-dimensional array with a specified size and data type.
2. **Initialization:** Initialize array elements during declaration or assign values after declaration.
3. **Accessing Elements:** Access individual elements using their index (starting from 0).
4. **Iteration:** Use loops (e.g., `for` loop) to iterate through array elements for input, processing, or output.

Source Code

```
#include <stdio.h> // Required for input/output functions

int main() {
    // Declare and initialize a one-dimensional integer array
    int numbers[5] = {10, 20, 30, 40, 50};
    int i; // Loop counter

    printf("Elements of the array:\n");
    // Iterate through the array and print each element
    for (i = 0; i < 5; i++) {
        printf("Element at index %d: %d\n", i, numbers[i]);
    }

    // Modify an element
    numbers[2] = 35; // Change the value at index 2

    printf("\nArray after modifying element at index 2:\n");
    for (i = 0; i < 5; i++) {
        printf("Element at index %d: %d\n", i, numbers[i]);
    }

    // Calculate the sum of array elements
    int sum = 0;
    for (i = 0; i < 5; i++) {
        sum += numbers[i];
    }
    printf("\nSum of all elements: %d\n", sum);

    return 0; // Indicate successful execution
}
```

Input

No explicit input is required for this program.

Expected Output

```
Elements of the array:
```

```
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50

Array after modifying element at index 2:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 35
Element at index 3: 40
Element at index 4: 50

Sum of all elements: 145
```

Lab 7: Arrays : Multi dimensional

Title

Working with Multi-Dimensional Arrays (Matrices)

Aim

To understand how to declare, initialize, and access elements of a multi-dimensional array (e.g., 2D array or matrix) in C.

Procedure

1. **Declaration:** Declare a multi-dimensional array with specified dimensions.
2. **Initialization:** Initialize array elements during declaration or assign values after declaration using nested loops.
3. **Accessing Elements:** Access individual elements using multiple indices (e.g., `array[row][column]`).
4. **Iteration:** Use nested loops to iterate through all elements for input, processing, or output.

Source Code

```
#include <stdio.h> // Required for input/output functions

int main() {
    // Declare and initialize a 2x3 integer matrix
    int matrix[2][3] = {
        {1, 2, 3},  // Row 0
        {4, 5, 6}   // Row 1
    };
    int i, j; // Loop counters for rows and columns

    printf("Elements of the 2x3 matrix:\n");
    // Iterate through rows
    for (i = 0; i < 2; i++) {
        // Iterate through columns
        for (j = 0; j < 3; j++) {
            printf("%d ", matrix[i][j]); // Print element followed by a space
        }
        printf("\n"); // Move to the next line after each row
    }

    // Access a specific element
    printf("\nElement at matrix[1][1]: %d\n", matrix[1][1]); // Should be 5

    // Modify an element
    matrix[0][0] = 10; // Change the value at row 0, column 0

    printf("\nMatrix after modifying element at [0][0]:\n");
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 3; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    return 0; // Indicate successful execution
}
```

Input

No explicit input is required for this program.

Expected Output

```
Elements of the 2x3 matrix:
1 2 3
4 5 6

Element at matrix[1][1]: 5

Matrix after modifying element at [0][0]:
10 2 3
4 5 6
```

Lab 8: Strings, structures and union

Title

Manipulating Strings, Structures, and Unions

Aim

To understand and implement strings, structures, and unions in C programming for handling complex data.

Procedure

1. **Strings:**
   - Declare and initialize character arrays to store strings.
   - Use string manipulation functions from `string.h` (e.g., `strcpy()`, `strcat()`, `strlen()`, `strcmp()`).
2. **Structures:**
   - Define a structure to group related data items of different data types.
   - Declare structure variables and access their members using the dot (`.`) operator.
3. **Unions:**
   - Define a union to store different data types in the same memory location.
   - Understand that only one member of a union can hold a value at any given time.

Source Code

```
#include <stdio.h>  // Required for input/output functions
#include <string.h> // Required for string manipulation functions

// --- Structure Definition ---
// Define a structure named 'Student'
struct Student {
    int roll_no;
    char name[50];
    float marks;
};

// --- Union Definition ---
// Define a union named 'Data'
union Data {
```

```c
    int i;
    float f;
    char str[20];
};

int main() {
    // --- String Example ---
    char greeting[20] = "Hello";
    char name[20] = "World";
    char combined[40]; // Buffer for concatenated string

    printf("--- String Operations ---\n");
    printf("Original greeting: %s\n", greeting);
    printf("Length of greeting: %zu\n", strlen(greeting)); // %zu for size_t

    // Copy "C Programming" to greeting (be careful with buffer size)
    strcpy(greeting, "C Programming");
    printf("New greeting after strcpy: %s\n", greeting);

    // Concatenate greeting and name into combined
    strcpy(combined, greeting); // Copy greeting first
    strcat(combined, " ");      // Add a space
    strcat(combined, name);     // Add name
    printf("Combined string: %s\n", combined);

    // Compare two strings
    char s1[] = "apple";
    char s2[] = "banana";
    char s3[] = "apple";
    printf("Comparison (s1 vs s2): %d (0 if equal, <0 if s1<s2, >0 if
s1>s2)\n", strcmp(s1, s2));
    printf("Comparison (s1 vs s3): %d\n", strcmp(s1, s3));

    // --- Structure Example ---
    // Declare a structure variable
    struct Student s1_data;

    printf("\n--- Structure Example ---\n");
    // Assign values to structure members
    s1_data.roll_no = 101;
    strcpy(s1_data.name, "John Doe"); // Copy string into name member
    s1_data.marks = 85.5;

    // Access and print structure members
    printf("Student Roll No: %d\n", s1_data.roll_no);
    printf("Student Name: %s\n", s1_data.name);
    printf("Student Marks: %.2f\n", s1_data.marks);

    // --- Union Example ---
    union Data data_union;

    printf("\n--- Union Example ---\n");
    // Assign an integer value
    data_union.i = 10;
    printf("data_union.i: %d\n", data_union.i);
    printf("data_union.f: %f (garbage value)\n", data_union.f); // f will be
garbage
    printf("data_union.str: %s (garbage value)\n", data_union.str); // str
will be garbage
    printf("Size of union Data: %zu bytes\n", sizeof(union Data)); // Size is
max of its members

    // Assign a float value (overwrites previous integer value)
    data_union.f = 22.5;
    printf("data_union.f: %.2f\n", data_union.f);
```

```
    printf("data_union.i: %d (garbage value)\n", data_union.i); // i will be
garbage

    // Assign a string value (overwrites previous float value)
    strcpy(data_union.str, "Hello Union");
    printf("data_union.str: %s\n", data_union.str);
    printf("data_union.f: %f (garbage value)\n", data_union.f); // f will be
garbage

    return 0; // Indicate successful execution
}
```

Input

No explicit input is required for this program.

Expected Output

```
--- String Operations ---
Original greeting: Hello
Length of greeting: 5
New greeting after strcpy: C Programming
Combined string: C Programming World
Comparison (s1 vs s2): -1 (0 if equal, <0 if s1<s2, >0 if s1>s2)
Comparison (s1 vs s3): 0

--- Structure Example ---
Student Roll No: 101
Student Name: John Doe
Student Marks: 85.50

--- Union Example ---
data_union.i: 10
data_union.f: 0.000000 (garbage value)
data_union.str:  (garbage value)
Size of union Data: 20 bytes
data_union.f: 22.50
data_union.i: 1075314688 (garbage value)
data_union.str: Hello Union
data_union.f: 1.700000 (garbage value)
```

(Note: The garbage values printed for union members not currently active will vary.)

Lab 9: Functions

Title

Implementing User-Defined Functions

Aim

To understand the concept of functions in C, including declaration, definition, and calling, and to implement user-defined functions for modular programming.

Procedure

1. **Function Prototype:** Declare the function prototype (signature) before `main()`.
2. **Function Definition:** Define the function's body (implementation) after `main()` or before its first call.
3. **Function Call:** Call the function from `main()` or another function, passing arguments if required.
4. **Return Value:** Understand how functions return values.

Source Code

```
#include <stdio.h> // Required for input/output functions

// Function Prototype (Declaration)
// This tells the compiler about the function's return type, name, and
parameters.
int add(int a, int b);
void greet(char name[]); // Function that doesn't return a value

int main() {
    int num1 = 10, num2 = 5;
    int result;

    // Call the 'add' function
    result = add(num1, num2);
    printf("Sum of %d and %d is: %d\n", num1, num2, result);

    // Call the 'greet' function
    greet("Alice");
    greet("Bob");

    return 0; // Indicate successful execution
}

// Function Definition (Implementation)
// This is where the actual code for the function resides.

// Function to add two integers and return their sum
int add(int a, int b) {
    return a + b; // Returns the sum of a and b
}

// Function to print a greeting message
void greet(char name[]) {
    printf("Hello, %s! Welcome to the function world.\n", name);
}
```

Input

No explicit input is required for this program.

Expected Output

```
Sum of 10 and 5 is: 15
Hello, Alice! Welcome to the function world.
Hello, Bob! Welcome to the function world.
```

Lab 10: Functions

Title

Advanced Function Concepts: Pass by Value and Pass by Reference

Aim

To differentiate between pass by value and pass by reference in C functions and understand their implications on argument passing.

Procedure

1. **Pass by Value:**
   - Create a function that takes arguments by value.
   - Observe that changes made to parameters inside the function do not affect the original variables in the calling function.
2. **Pass by Reference:**
   - Create a function that takes arguments by reference (using pointers).
   - Observe that changes made to parameters inside the function *do* affect the original variables in the calling function.
3. **Demonstrate:** Show how to swap two numbers using both methods (or explain why pass by value won't work for swapping).

Source Code

```
#include <stdio.h> // Required for input/output functions

// Function demonstrating Pass by Value
// 'x' and 'y' are copies of 'a' and 'b' from main.
// Changes to 'x' and 'y' here will not affect 'a' and 'b' in main.
void passByValue(int x, int y) {
    printf("Inside passByValue function (before change):\n");
    printf("x = %d, y = %d\n", x, y);

    x = 100; // Change x
    y = 200; // Change y

    printf("Inside passByValue function (after change):\n");
    printf("x = %d, y = %d\n", x, y);
}

// Function demonstrating Pass by Reference
// 'ptrX' and 'ptrY' are pointers to 'a' and 'b' from main.
// Changes to *ptrX and *ptrY will affect 'a' and 'b' in main.
void passByReference(int *ptrX, int *ptrY) {
    printf("Inside passByReference function (before change):\n");
    printf("*ptrX = %d, *ptrY = %d\n", *ptrX, *ptrY);

    *ptrX = 1000; // Change the value at the address ptrX points to
    *ptrY = 2000; // Change the value at the address ptrY points to

    printf("Inside passByReference function (after change):\n");
    printf("*ptrX = %d, *ptrY = %d\n", *ptrX, *ptrY);
}

int main() {
    int a = 10, b = 20;

    printf("--- Demonstrating Pass by Value ---\n");
```

```
    printf("Before calling passByValue: a = %d, b = %d\n", a, b);
    passByValue(a, b); // Pass copies of a and b
    printf("After calling passByValue: a = %d, b = %d\n", a, b); // a and b
remain unchanged

    printf("\n--- Demonstrating Pass by Reference ---\n");
    printf("Before calling passByReference: a = %d, b = %d\n", a, b);
    passByReference(&a, &b); // Pass the memory addresses of a and b
    printf("After calling passByReference: a = %d, b = %d\n", a, b); // a and
b are changed

    return 0; // Indicate successful execution
}
```

Input

No explicit input is required for this program.

Expected Output

```
--- Demonstrating Pass by Value ---
Before calling passByValue: a = 10, b = 20
Inside passByValue function (before change):
x = 10, y = 20
Inside passByValue function (after change):
x = 100, y = 200
After calling passByValue: a = 10, b = 20

--- Demonstrating Pass by Reference ---
Before calling passByReference: a = 10, b = 20
Inside passByReference function (before change):
*ptrX = 10, *ptrY = 20
Inside passByReference function (after change):
*ptrX = 1000, *ptrY = 2000
After calling passByReference: a = 1000, b = 2000
```

Lab 11: Pointers Lab

Title

Introduction to Pointers and Pointer Arithmetic

Aim

To understand the concept of pointers, how to declare and initialize them, and perform basic pointer arithmetic.

Procedure

1. **Declare Pointers:** Declare pointer variables using the `*` operator.
2. **Initialize Pointers:** Initialize pointers with the address of another variable using the `&` operator.
3. **Dereference Pointers:** Access the value pointed to by a pointer using the `*` operator (dereferencing).
4. **Pointer Arithmetic:** Perform arithmetic operations (addition, subtraction) on pointers and observe their effect on memory addresses.

Source Code

```
#include <stdio.h> // Required for input/output functions

int main() {
    int var = 10;    // Declare an integer variable
    int *ptr;        // Declare an integer pointer

    // Assign the address of 'var' to 'ptr'
    ptr = &var;

    printf("--- Pointer Basics ---\n");
    printf("Value of var: %d\n", var);          // Output: 10
    printf("Address of var: %p\n", &var);       // Output: Memory address of
var
    printf("Value of ptr (address it holds): %p\n", ptr); // Output: Same as
address of var
    printf("Value pointed to by ptr (*ptr): %d\n", *ptr); // Output: 10
(dereferencing)

    // Change value using pointer
    *ptr = 20; // Change the value at the address ptr points to
    printf("\nAfter changing value via pointer:\n");
    printf("Value of var: %d\n", var);          // Output: 20
    printf("Value pointed to by ptr (*ptr): %d\n", *ptr); // Output: 20

    // --- Pointer Arithmetic ---
    int arr[5] = {10, 20, 30, 40, 50};
    int *arr_ptr = arr; // arr_ptr points to the first element (arr[0])

    printf("\n--- Pointer Arithmetic ---\n");
    printf("Value of arr_ptr (address of arr[0]): %p\n", arr_ptr);
    printf("Value pointed by arr_ptr (*arr_ptr): %d\n", *arr_ptr); // Output:
10

    arr_ptr++; // Increment pointer to point to the next integer (arr[1])
    printf("Value of arr_ptr after increment: %p\n", arr_ptr);
    printf("Value pointed by arr_ptr (*arr_ptr): %d\n", *arr_ptr); // Output:
20
```

```
    arr_ptr = arr_ptr + 2; // Increment pointer by 2 positions (points to
arr[3])
    printf("Value of arr_ptr after adding 2: %p\n", arr_ptr);
    printf("Value pointed by arr_ptr (*arr_ptr): %d\n", *arr_ptr); // Output:
40

    return 0; // Indicate successful execution
}
```

Input

No explicit input is required for this program.

Expected Output

```
--- Pointer Basics ---
Value of var: 10
Address of var: 0x7ffc7b7a0c04 (example address)
Value of ptr (address it holds): 0x7ffc7b7a0c04 (example address)
Value pointed to by ptr (*ptr): 10

After changing value via pointer:
Value of var: 20
Value pointed to by ptr (*ptr): 20

--- Pointer Arithmetic ---
Value of arr_ptr (address of arr[0]): 0x7ffc7b7a0c10 (example address)
Value pointed by arr_ptr (*arr_ptr): 10
Value of arr_ptr after increment: 0x7ffc7b7a0c14 (example address, +4 bytes
for int)
Value pointed by arr_ptr (*arr_ptr): 20
Value of arr_ptr after adding 2: 0x7ffc7b7a0c1c (example address, +8 bytes
for 2 ints)
Value pointed by arr_ptr (*arr_ptr): 40
```

(Note: Memory addresses will vary each time the program is run.)

Lab 12: Pointers

Title

Pointers and Arrays, Pointers to Functions

Aim

To understand the relationship between pointers and arrays, and how to use pointers to functions.

Procedure

1. **Pointers and Arrays:**
   - Demonstrate that an array name can act as a pointer to its first element.
   - Access array elements using pointer arithmetic.
2. **Pointers to Functions:**
   - Declare a pointer that can point to a function.
   - Assign the address of a function to the function pointer.
   - Call the function using the function pointer.

Source Code

```
#include <stdio.h> // Required for input/output functions

// --- Functions for Function Pointers ---
int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

int main() {
    // --- Pointers and Arrays ---
    int numbers[5] = {10, 20, 30, 40, 50};
    int *ptr_to_array = numbers; // ptr_to_array points to the first element
of numbers

    printf("--- Pointers and Arrays ---\n");
    printf("Using array name as pointer: %d\n", *numbers); // Dereference
array name
    printf("Using pointer to array: %d\n", *ptr_to_array);

    printf("Accessing array elements using pointer arithmetic:\n");
    for (int i = 0; i < 5; i++) {
        // (ptr_to_array + i) gives the address of the i-th element
        // *(ptr_to_array + i) dereferences it to get the value
        printf("Element %d: %d\n", i, *(ptr_to_array + i));
    }

    // --- Pointers to Functions ---
    // Declare a function pointer that takes two integers and returns an
integer
    int (*func_ptr)(int, int);

    printf("\n--- Pointers to Functions ---\n");

    // Assign the address of the 'add' function to func_ptr
```

```
    func_ptr = &add; // Or simply func_ptr = add; (function name decays to
pointer)
    printf("Calling add function via pointer: %d\n", func_ptr(15, 7));

    // Assign the address of the 'subtract' function to func_ptr
    func_ptr = &subtract;
    printf("Calling subtract function via pointer: %d\n", func_ptr(15, 7));

    return 0; // Indicate successful execution
}
```

Input

No explicit input is required for this program.

Expected Output

```
--- Pointers and Arrays ---
Using array name as pointer: 10
Using pointer to array: 10
Accessing array elements using pointer arithmetic:
Element 0: 10
Element 1: 20
Element 2: 30
Element 3: 40
Element 4: 50

--- Pointers to Functions ---
Calling add function via pointer: 22
Calling subtract function via pointer: 8
```

Lab 13: File: reading and writing

Title

Basic File I/O: Reading and Writing Text Files

Aim

To learn how to open, read from, and write to text files in C using `fopen()`, `fprintf()`, `fscanf()`, and `fclose()`.

Procedure

1. **Open File:** Use `fopen()` to open a file in a specific mode (`"w"` for write, `"r"` for read, `"a"` for append). Check if the file was opened successfully.
2. **Write to File:** Use `fprintf()` to write formatted data to the file.
3. **Close File (Write):** Use `fclose()` to close the file after writing.
4. **Open File (Read):** Reopen the same file in read mode (`"r"`).
5. **Read from File:** Use `fscanf()` to read formatted data from the file.
6. **Close File (Read):** Use `fclose()` to close the file after reading.
7. **Error Handling:** Include checks for `NULL` file pointers to handle file opening errors.

Source Code

```
#include <stdio.h> // Required for file I/O functions

int main() {
    FILE *file_ptr; // Declare a file pointer

    // --- Writing to a file ---
    // Open "example.txt" in write mode ("w")
    // If the file doesn't exist, it will be created. If it exists, its
content will be truncated.
    file_ptr = fopen("example.txt", "w");

    // Check if file was opened successfully
    if (file_ptr == NULL) {
        printf("Error opening file for writing!\n");
        return 1; // Indicate an error
    }

    printf("Writing to file...\n");
    // Write formatted data to the file
    fprintf(file_ptr, "Hello, this is a line written to the file.\n");
    fprintf(file_ptr, "This is the second line with a number: %d\n", 123);
    fprintf(file_ptr, "And a float value: %.2f\n", 45.67);

    // Close the file
    fclose(file_ptr);
    printf("File written successfully.\n");

    // --- Reading from a file ---
    // Open "example.txt" in read mode ("r")
    file_ptr = fopen("example.txt", "r");

    // Check if file was opened successfully
    if (file_ptr == NULL) {
        printf("Error opening file for reading!\n");
        return 1; // Indicate an error
    }
```

```
    printf("\nReading from file...\n");
    char buffer[100]; // Buffer to store read lines
    int num_read;
    float float_read;

    // Read and print the first line
    // fgets reads a line including newline, up to (size-1) characters
    if (fgets(buffer, sizeof(buffer), file_ptr) != NULL) {
        printf("%s", buffer);
    }

    // Read the second line which contains a number
    // fscanf reads formatted input
    if (fscanf(file_ptr, "This is the second line with a number: %d\n",
&num_read) == 1) {
        printf("Second line (number): This is the second line with a number:
%d\n", num_read);
    } else {
        printf("Error reading number from file.\n");
    }

    // Read the third line which contains a float
    if (fscanf(file_ptr, "And a float value: %f\n", &float_read) == 1) {
        printf("Third line (float): And a float value: %.2f\n", float_read);
    } else {
        printf("Error reading float from file.\n");
    }

    // Close the file
    fclose(file_ptr);
    printf("File read successfully.\n");

    return 0; // Indicate successful execution
}
```

Input

No explicit input is required for this program. It creates and reads from `example.txt`.

Expected Output

(Content of `example.txt` *after writing:)*

```
Hello, this is a line written to the file.
This is the second line with a number: 123
And a float value: 45.67
```

(Output to console:)

```
Writing to file...
File written successfully.

Reading from file...
Hello, this is a line written to the file.
Second line (number): This is the second line with a number: 123
Third line (float): And a float value: 45.67
File read successfully.
```

Lab 14: File Handling `fputw()`, `fgetw()`

Title

File Handling with `fputc()` and `fgetc()` (Character I/O)

Aim

To understand and implement character-by-character file input/output using `fputc()` for writing and `fgetc()` for reading.

Procedure

1. **Open File (Write):** Open a file in write mode (`"w"`).
2. **Write Characters:** Use `fputc()` in a loop to write individual characters to the file.
3. **Close File (Write):** Close the file.
4. **Open File (Read):** Open the same file in read mode (`"r"`).
5. **Read Characters:** Use `fgetc()` in a loop to read individual characters until the end of the file (`EOF`) is reached.
6. **Close File (Read):** Close the file.
7. **Error Handling:** Include checks for `NULL` file pointers.

Source Code

```
#include <stdio.h> // Required for file I/O functions

int main() {
    FILE *file_ptr;
    char ch;

    // --- Writing characters to a file ---
    file_ptr = fopen("char_example.txt", "w");
    if (file_ptr == NULL) {
        printf("Error opening file for writing!\n");
        return 1;
    }

    printf("Writing characters to file...\n");
    char *message = "Hello, C File Handling!";
    for (int i = 0; message[i] != '\0'; i++) {
        fputc(message[i], file_ptr); // Write one character at a time
    }
    fputc('\n', file_ptr); // Add a newline for readability
    fclose(file_ptr);
    printf("Characters written successfully.\n");

    // --- Reading characters from a file ---
    file_ptr = fopen("char_example.txt", "r");
    if (file_ptr == NULL) {
        printf("Error opening file for reading!\n");
        return 1;
    }

    printf("\nReading characters from file:\n");
    // Read characters one by one until End Of File (EOF) is reached
    while ((ch = fgetc(file_ptr)) != EOF) {
        printf("%c", ch); // Print the character
    }
    printf("\n"); // Add a newline after reading all characters
```

```
    fclose(file_ptr);
    printf("Characters read successfully.\n");

    return 0;
}
```

Input

No explicit input is required for this program. It creates and reads from `char_example.txt`.

Expected Output

(Content of `char_example.txt` after writing:)

```
Hello, C File Handling!
```

(Output to console:)

```
Writing characters to file...
Characters written successfully.

Reading characters from file:
Hello, C File Handling!

Characters read successfully.
```

Lab 15: Creating Macros

Title

Defining and Using Macros in C

Aim

To understand the concept of preprocessor macros in C and how to define and use them for constant definitions and simple function-like substitutions.

Procedure

1. **Define Simple Macros:** Use `#define` to define symbolic constants.
2. **Define Function-like Macros:** Use `#define` to create macros that take arguments, similar to functions.
3. **Use Macros:** Incorporate the defined macros into the C code.
4. **Observe Preprocessing:** Understand that macros are expanded by the preprocessor before compilation.
5. **Caution:** Be aware of potential pitfalls with function-like macros (e.g., side effects, operator precedence).

Source Code

```
#include <stdio.h> // Required for input/output functions

// --- Simple Macros (Symbolic Constants) ---
#define PI 3.14159 // Define PI as a constant
#define MAX_VALUE 100 // Define a maximum value

// --- Function-like Macros ---
// Macro to find the maximum of two numbers
// IMPORTANT: Use parentheses around arguments and the entire expression
// to avoid issues with operator precedence during expansion.
#define MAX(a, b) ((a) > (b) ? (a) : (b))

// Macro to calculate the area of a circle
#define AREA_CIRCLE(radius) (PI * (radius) * (radius))

int main() {
    // --- Using Simple Macros ---
    int radius = 5;
    float area = AREA_CIRCLE(radius); // Macro expansion: (3.14159 * (5) *
(5))

    printf("--- Using Macros ---\n");
    printf("Value of PI: %.5f\n", PI);
    printf("Maximum allowed value: %d\n", MAX_VALUE);
    printf("Area of circle with radius %d: %.2f\n", radius, area);

    // --- Using Function-like Macros ---
    int x = 10, y = 20;
    int max_num = MAX(x, y); // Macro expansion: ((10) > (20) ? (10) : (20))

    printf("Maximum of %d and %d is: %d\n", x, y, max_num);

    // Demonstrate a potential pitfall if parentheses are not used in MAX
macro:
    // If MAX was #define MAX(a, b) a > b ? a : b
```

```
    // Then MAX(x++, y) would expand to x++ > y ? x++ : y, causing x to
increment twice if x > y
    // With parentheses: ((x++) > (y) ? (x++) : (y)) - still has side
effects, but less problematic.
    // It's generally safer to use inline functions for complex logic or when
side effects are a concern.
    int p = 5, q = 10;
    printf("MAX(p++, q): %d (p becomes %d)\n", MAX(p++, q), p); // p
increments once due to (p) in macro

    return 0; // Indicate successful execution
}
```

Input

No explicit input is required for this program.

Expected Output

```
--- Using Macros ---
Value of PI: 3.14159
Maximum allowed value: 100
Area of circle with radius 5: 78.54

Maximum of 10 and 20 is: 20
MAX(p++, q): 10 (p becomes 6)
```

(Note: The output for `MAX(p++, q)` demonstrates that `p` increments only once because `p` is evaluated as `(p)` within the macro due to the parentheses, and then `p++` is evaluated inside the ternary operator. Without the parentheses, `p` might increment twice depending on the expression.)