

Course Name:	Programming in C	Semester:	II
Date of Performance:	26/03/2025	DIV/ Batch No:	c2-2
Student Name:	Ashwera Hasan	Roll No:	16010124107

Experiment No: 6
Title: User defined functions

Aim and Objective of the Experiment:
Write a program in C to implement user defined functions.

COs to be achieved:
CO4: Design modular programs using functions and the use of structure and union.

Theory:
<p style="text-align: center;">Introduction to User-Defined Function</p> <p>In C programming, a User-Defined Function (UDF) is a function that is defined by the programmer to perform a specific task. Functions are essential tools for modularizing a program, allowing complex tasks to be divided into smaller, more manageable chunks. User-defined functions enhance code reusability, maintainability, and readability by encapsulating specific functionality into independent units.</p> <p>In C, a function is defined once and can be called multiple times within a program. This allows for more organized, efficient, and error-free code.</p> <p>A User-Defined Function in C consists of the following parts:</p> <ul style="list-style-type: none"> • Function Declaration/Prototype (Optional but recommended) • Function Definition • Function Call <p>Function Declaration/Prototype: The function prototype is a declaration of the function that specifies the function name, return type, and the types of parameters. It is often placed before the main() function to inform the compiler about the function's characteristics, which helps in type checking during compilation.</p> <p>Syntax:</p> <pre>return_type function_name(parameter_type1, parameter_type2, ...);</pre> <p>Example:</p> <pre>int add(int a, int b); // Function prototype</pre>

Function Definition: This is where the function is actually defined. It includes the function's body, where the desired operations are performed. The function definition must match the declaration or prototype.

Syntax:

```
return_type function_name(parameter1, parameter2, ...) {  
    // Body of the function  
    // Perform operations  
    return result; // Optional, if return type is not void  
}
```

Example:

```
int add(int a, int b) {  
    return a + b; // Adds the two integers and returns the result  
}
```

Function Call: This is where the function is invoked in the program. To call a function, you simply write its name followed by parentheses, passing the necessary arguments (if any).

Example:

```
int sum = add(5, 3); // Calling the 'add' function with arguments 5 and 3
```

Function Types in C

Functions with Return Values: These functions perform a task and return a value. The return type is specified in the function prototype and definition. For example, the add() function in the previous example returns an integer.

Functions without Return Values (void functions): These functions do not return any value. The return type in the function prototype and definition is void. Such functions are typically used to perform actions like printing messages or modifying global variables.

Example:

```
void print_hello() {  
    printf("Hello, World!\n");  
}
```

Functions with Arguments: These functions take arguments as input, which are used in the body of the function to perform a specific task. Arguments can be passed by value or by reference (using pointers).

Example:

```
int multiply(int a, int b) {  
    return a * b; // Multiplies two integers and returns the result  
}
```

Functions without Arguments: These functions do not take any arguments, but they may still perform a task like printing output or modifying data.

Example:

```
void display_message() {  
    printf("This is a user-defined function without arguments.\n");  
}
```

Benefits of Using User-Defined Functions in C:

- **Reusability:** Once a function is defined, it can be called multiple times in different parts of the program, reducing redundancy and enhancing code reusability.
- **Modularity:** Functions allow the programmer to divide the program into smaller, more manageable sections, making the code easier to read, debug, and maintain.
- **Improved Readability:** By using functions, the program becomes more organized. Instead of having one large block of code, the program can be divided into smaller sections, each performing a specific task.
- **Easier Debugging and Testing:** Since each function is focused on a single task, testing and debugging become easier. If there is an issue with the function, you can focus on fixing that specific function without impacting the rest of the program.
- **Abstraction:** Functions abstract away the details of the implementation. This means that the main part of the program only needs to know what the function does, not how it does it.
- **Better Maintenance:** If a function needs to be updated or modified, you can change the function definition in one place, and all calls to that function throughout the program will automatically use the updated version.

Problem Statements:

1. Write a C program to find the mean, median and mode of an array of numbers using a user-defined function.
2. Write a C program that multiplies two matrices. The function should take input for the dimensions of two matrices and their elements, then compute and display the product of the two matrices using functions.

Code :

```
ere x project.c x
1  #include <stdio.h>
2
3  void sortArray(int arr[], int n);
4  double findMean(int arr[], int n);
5  double findMedian(int arr[], int n);
6  int findMode(int arr[], int n);
7
8  int main() {
9      int n;
10     printf("Enter number of elements: ");
11     scanf("%d", &n);
12
13     int arr[n];
14     printf("Enter elements: ");
15     for(int i = 0; i < n; i++) {
16         scanf("%d", &arr[i]);
17     }
18
19     printf("Mean: %.2f\n", findMean(arr, n));
20     printf("Median: %.2f\n", findMedian(arr, n));
21     printf("Mode: %d\n", findMode(arr, n));
22
23     return 0;
24 }
```

1.

```
        return 0;
    }

    void sortArray(int arr[], int n) {
        for(int i = 0; i < n - 1; i++) {
            for(int j = 0; j < n - i - 1; j++) {
                if(arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }

    double findMean(int arr[], int n) {
        double sum = 0;
        for(int i = 0; i < n; i++) {
            sum += arr[i];
        }
        return sum / n;
    }
}
```

```
-- project --  
double findMedian(int arr[], int n) {  
    sortArray(arr, n);  
    if(n % 2 == 0) {  
        return (arr[n/2 - 1] + arr[n/2]) / 2.0;  
    } else {  
        return arr[n/2];  
    }  
}  
  
int findMode(int arr[], int n) {  
    sortArray(arr, n);  
    int mode = arr[0], maxCount = 1, count = 1;  
    for(int i = 1; i < n; i++) {  
        if(arr[i] == arr[i - 1]) {  
            count++;  
        } else {  
            count = 1;  
        }  
        if(count > maxCount) {  
            maxCount = count;  
            mode = arr[i];  
        }  
    }  
    return mode;  
}
```

2.

```
#include <stdio.h>

void multiplyMatrices(int first[][10], int second[][10], int result[][10], int r1, int c1, int r2, int c2)
void inputMatrix(int matrix[][10], int row, int col);
void displayMatrix(int matrix[][10], int row, int col);

int main() {
    int r1, c1, r2, c2;
    printf("Enter rows and columns for first matrix: ");
    scanf("%d %d", &r1, &c1);
    printf("Enter rows and columns for second matrix: ");
    scanf("%d %d", &r2, &c2);

    if (c1 != r2) {
        printf("Matrix multiplication not possible.\n");
        return 0;
    }

    int first[10][10], second[10][10], result[10][10];

    printf("Enter elements of first matrix:\n");
    inputMatrix(first, r1, c1);
    printf("Enter elements of second matrix:\n");
    inputMatrix(second, r2, c2);

    multiplyMatrices(first, second, result, r1, c1, r2, c2);

    printf("Product of matrices:\n");
    displayMatrix(result, r1, c2);

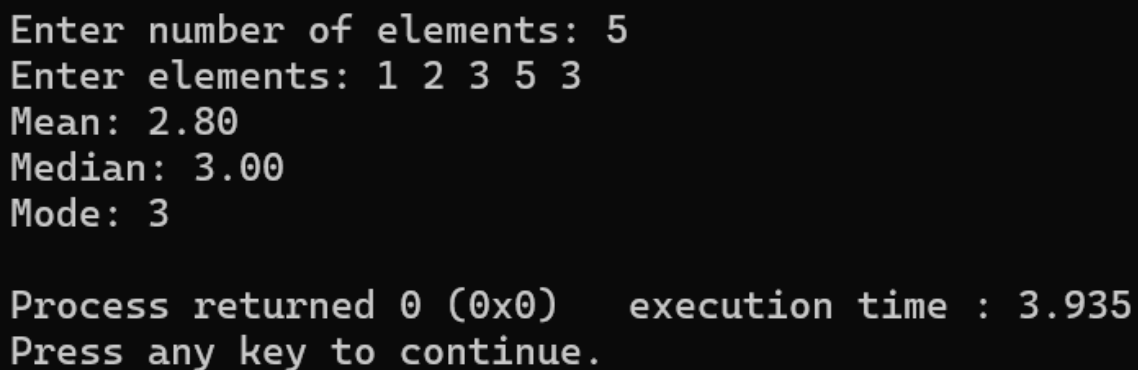
    return 0;
}

void inputMatrix(int matrix[][10], int row, int col) {
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }
}

void multiplyMatrices(int first[][10], int second[][10], int result[][10], int r1, int c1, int r2, int c2) {
    for (int i = 0; i < r1; i++) {
        for (int j = 0; j < c2; j++) {
            result[i][j] = 0;
            for (int k = 0; k < c1; k++) {
                result[i][j] += first[i][k] * second[k][j];
            }
        }
    }
}
```

```
}  
  
void multiplyMatrices(int first[][10], int second[][10], int result[][10], int r1, int c1, int c2)  
{  
    for (int i = 0; i < r1; i++) {  
        for (int j = 0; j < c2; j++) {  
            result[i][j] = 0;  
            for (int k = 0; k < c1; k++) {  
                result[i][j] += first[i][k] * second[k][j];  
            }  
        }  
    }  
}  
  
void displayMatrix(int matrix[][10], int row, int col) {  
    for (int i = 0; i < row; i++) {  
        for (int j = 0; j < col; j++) {  
            printf("%d ", matrix[i][j]);  
        }  
        printf("\n");  
    }  
}
```

Output:



```
Enter number of elements: 5  
Enter elements: 1 2 3 5 3  
Mean: 2.80  
Median: 3.00  
Mode: 3  
  
Process returned 0 (0x0)   execution time : 3.935  
Press any key to continue.  
|
```

1.


```
C:\Users\syeda\OneDrive\Doc  X  +  v

Enter rows and columns for first matrix: 2 2
Enter rows and columns for second matrix: 2 2
Enter elements of first matrix:
1 2 3 4
Enter elements of second matrix:
1 2 3 4
Product of matrices:
7 10
15 22

Process returned 0 (0x0)    execution time : 7.556 s
Press any key to continue.
|
```

2.

Post Lab Subjective/Objective type Questions:

1. What is the difference between Call by Value and Call by Address?

Call by Value means passing a copy of a variable to a function. Any changes inside the function do not affect the original variable. Since a copy is created, it requires extra memory.

In contrast, Call by Address passes the memory address of a variable to a function. Changes made inside the function directly affect the original variable. This method is useful for handling arrays and large data structures efficiently. It avoids copying values, making it more memory-efficient. Call by Address uses pointers to access variables.

2. Explain recursion using functions in C with an example. (Handwritten)

Recursion means a function calling itself to solve a problem repeatedly till the base condition is fulfilled. This works slightly similar to loops, except here, the function itself is called within it.

CODE:-

```
int factorial (int n)
{
    if (n == 0) return 1;
    return n * factorial (n-1);
}

int main () { int num = 5;
    printf ("Factorial of %d is %d", num,
    factorial (num));
    return 0; }
```

Here, the same function is called again and again for 5, 4, 3, 2, 1 and stops at 0 due to base case condition.

Conclusion:

We learned that Call by Value and Call by Address affect how functions handle data. In the mean, median, and mode program, arrays were passed by reference for sorting efficiency. In matrix multiplication, matrices were passed by reference to modify results directly. Using the right method improves performance and prevents unnecessary memory usage. Understanding these concepts helps write efficient and error-free programs.

Signature of faculty in-charge with Date: