Course Name:	Programming in C	Semester:	II
Date of Performance:	24/02/2025	DIV/ Batch No:	[redacted]
Student Name:	MAYURI MANOJ KUMBHAR	Roll No:	[redacted]

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:

Introduction to User-Defined Function

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.

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.

A User-Defined Function in C consists of the following parts:

- Function Declaration/Prototype (Optional but recommended)
- Function Definition
- Function Call

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.

Syntax:

```
return_type function_name(parameter_type1, parameter_type2, ...);
```

Example:

```
int add(int a, int b);  // Function prototype
```

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 :

Question 1.]

```
#include <stdio.h>
void sortArray(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n; i++)
      {
      for (j = 0; j < n - 1; j++)
      {
         if (arr[j] > arr[j + 1])
         {
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
         }
      }
   }
}
float mean(int arr[], int n)
{
   int i, sum = 0;
   for (i = 0; i < n; i++)
      {
      sum += arr[i];
   }
   return (float)sum / n;
}
float median(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 calculateMode(int arr[], int n) {
   int i, j, max = 0, count, mode = -1;
   for (i = 0; i < n; i++)
```

```
        {
        count = 1;
        for (j = i + 1; j < n; j++)
        {
           if (arr[j] == arr[i])
            {
              count++;
           }
        }
        if (count > max) {
           max = count;
           mode = arr[i];
        }
    }
    if (max == 1) {
        return -1;
    }
    return mode;
}
int main() {
    int n, i;
    printf("Enter no. of elements in array: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter elements: ");
    for (i = 0; i < n; i++)
        {
        scanf("%d", &arr[i]);
    }
    printf("Mean: %.2f\n", mean(arr, n));
    printf("Median: %.2f\n", median(arr, n));
    int mode = calculateMode(arr, n);
    if (mode == -1)
     {
        printf("No mode found\n");
    } else
        {
        printf("Mode = %d\n", mode);
    }
    return 0;
}
```

Question 2.]

```
#include <stdio.h>
void getmat(int matrix[][10], int rows, int cols);
void multiplymat(int firstMatrix[][10], int firstRows, int firstCols, int secondMatrix[][10], int secondRows, int secondCols, int resultMatrix[][10]);
void displaymat(int matrix[][10], int rows, int cols);
int main()
{
   int firstMatrix[10][10], secondMatrix[10][10], resultMatrix[10][10];
   int firstRows, firstCols, secondRows, secondCols;
   printf("Enter rows and columns for the first matrix: ");
   scanf("%d %d", &firstRows, &firstCols);
   printf("Enter rows and columns for the second matrix: ");
   scanf("%d %d", &secondRows, &secondCols);
   if (firstCols != secondRows)
      {
         printf("Matrix multiplication is not possible.\n");
   return -1;
   }
   printf("Enter elements of the first matrix:\n");
   getmat(firstMatrix, firstRows, firstCols);
   printf("Enter elements of the second matrix:\n");
   getmat(secondMatrix, secondRows, secondCols);
   multiplymat(firstMatrix, firstRows, firstCols, secondMatrix, secondRows, secondCols, resultMatrix);
   printf("Resultant Matrix:\n");
   displaymat(resultMatrix, firstRows, secondCols);
   return 0;
}
void getmat(int matrix[][10], int rows, int cols) {
   for (int i = 0; i < rows; i++)
      {
      for (int j = 0; j < cols; j++)
      {
         scanf("%d", &matrix[i][j]);
      }
   }
}
void multiplymat(int firstMatrix[][10], int firstRows, int firstCols, int secondMatrix[][10], int
```

```
secondRows, int secondCols, int resultMatrix[][10]) {
    for (int i = 0; i < firstRows; i++)
        {
        for (int j = 0; j < secondCols; j++)
        {
            resultMatrix[i][j] = 0;
            for (int k = 0; k < firstCols; k++)
            {
                resultMatrix[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
            }
        }
    }
}
void displaymat(int matrix[][10], int rows, int cols) {
    for (int i = 0; i < rows; i++)
        {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
```

Output:

Question 1.]

```
Enter no. of elements in array: 5
Enter elements: 1
2
3
4
4
Mean: 2.80
Median: 3.00
Mode = 4

Process returned 0 (0x0)   execution time : 24.270 s
Press any key to continue.
```

Question 2.]

```
Enter rows and columns for the first matrix: 2 2
Enter rows and columns for the second matrix: 2 2
Enter elements of the first matrix:
7 6
4 5
Enter elements of the second matrix:
11 15
21 10
Resultant Matrix:
203 165
149 110

Process returned 0 (0x0)   execution time : 18.507 s
Press any key to continue.
```

Post Lab Subjective/Objective type Questions:

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

Ans: **Call by value** *the actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn't affect actual parameters.*

Call by address *the address of actual arguments (or parameters) is passed to the formal parameters, which means any operation performed on formal parameters affects the value of actual parameters.*

2. Explain Recursion using Functions in C with an example. (Handwritten).

Ans:

Recursion is a process where a function calls itself directly or indirectly until a specific condition is met

→ Example: Factorial

```
#include<stdio.h>
int factorial(int n)
{
  if(n==0 || n==1) {
    return 1;
  } else {
    return n*factorial(n-1);
  }
}
int main() {
  int num;
printf("Enter a number:");
scanf("%d",&num);
printf("Factorial of %d = %d\n", num, factorial(num));
  return 0;
}
```

Conclusion:

In this experiment, we learnt about functions in C programming, its types, uses and application.
A function is a block of statements that performs a specific task.
Functions are used because of following reasons:
To improve the readability of code, improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch, debugging of the code would be easier if you use functions, as errors are easy to be traced, reduces the size of the code, duplicate set of statements are replaced by function calls.

Signature of faculty in-charge with Date: