|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Programming in C** | **Semester:** | **II** |
| **Date of Performance:** | **28 / 02 2025** | **DIV/ Batch No:** | **C4-1** |
| **Student Name:** | **Dhruv Pankhania** | **Roll No:** | **16010124216** |

**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 Argument**s: 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 :** |
| 1. #include<stdio.h>  *double* mean(*int* *arr[]*, *int* *size*);  *double* median (*int* *arr[]*, *int* *size*);  *int* mode (*int* *arr[]*, *int* *size*);  *int* sort(*int* *arr[]*, *int* *size*);  *int* main(){  *int* size;    printf("Enter the Size of the Array: ");    scanf("%d", &size);  *int* arr[size];    printf("Enter the Elements of the Array: ");    for(*int* i = 0; i<size; i++){    scanf("%d", &arr[i]);  }  printf("The Mean of the array is: %lf\n", mean(arr, size));  printf("The Median of the array is: %lf\n", median(arr, size));  printf("The Mode of the array is: %d\n", mode(arr, size));    return 0;  }  *double* mean(*int* *arr[]*, *int* *size*){  *int* sum = 0;    for(*int* i = 0; i<*size*; i++){      sum+= *arr*[i];    }    return (*double*)sum/*size*;  }  *double* median(*int* *arr[]*, *int* *size*){    sort(*arr*, *size*);  if(*size*%2){    return *arr*[(*size*-1)/2];  } else{    return *arr*[*size*/2];  }  }  *int* mode(*int* *arr[]*, *int* *size*){    sort(*arr*, *size*);  *int* maxFreq = 1, currentFreq = 1, modeValue = *arr*[0];    for(*int* i = 1; i<*size*; i++){      if(*arr*[i] == *arr*[i-1]){        currentFreq++;      } else{        currentFreq = 1;      }      if(currentFreq>maxFreq){        maxFreq = currentFreq;        modeValue = *arr*[i];      }    }    return modeValue;  }  *int* sort(*int* *arr[]*, *int* *size*){  for(*int* i = 0; i<*size*; i++){  for(*int* j = i+1; j<*size*; j++){  *int* temp = 0;  if(*arr*[i]>*arr*[j]){    temp = *arr*[i];  *arr*[i] = *arr*[j];  *arr*[j] = temp;  }  }  }  }  2.  #include<stdio.h>  void arrMultiplication(int size1i, int size1j, int arr1[size1i][size1j], int size2i, int size2j, int arr2[size2i][size2j], int result[size1i][size2j]);  int main(){  int size1i, size1j;  printf("Enter the size of the first 2D array you want to create: ");  scanf("%d %d", &size1i, &size1j);  int arr1[size1i][size1j];  printf("Enter the Numbers for the array: ");  for(int i = 0; i<size1i; i++){  for(int j = 0; j<size1j; j++){  scanf("%d", &arr1[i][j]);  }  }  int size2i, size2j;  printf("Enter the size of the second 2D array you want to create: ");  scanf("%d %d", &size2i, &size2j);  int arr2[size2i][size2j];  printf("Enter the Numbers for the array: ");  for(int i = 0; i<size2i; i++){  for(int j = 0; j<size2j; j++){  scanf("%d", &arr2[i][j]);  }  }  int result[size1i][size2j];  arrMultiplication(size1i, size1j, arr1, size2i, size2j, arr2, result);  for(int i = 0; i<size1i; i++){  for(int j = 0; j<size2j; j++){  printf("%d \t", result[i][j]);  }  printf("\n");  }  return 0;  }  void arrMultiplication(int size1i, int size1j, int arr1[size1i][size1j], int size2i, int size2j, int arr2[size2i][size2j], int result[size1i][size2j]){  if(size1j==size2i){  for(int i = 0; i<size1i; i++){  for(int j = 0; j<size2j; j++){  result[i][j]=0;  for(int k=0; k<size1j; k++){  result[i][j] += arr1[i][k]\*arr2[k][j];  }  }  }  } else{  printf("Array Multiplication Not Possible");  return;  }  } |

|  |
| --- |
| **Output:** |
| 1.   2. |

|  |
| --- |
| **Post Lab Subjective/Objective type Questions:** |
| 1. What is the difference between Call by Value and Call by Address?  **Call by Value** passes a copy of the actual value to a function, meaning changes inside the function do not affect the original variable. It ensures data security but requires extra memory for copying. **Call by Address** passes the memory address of the variable, allowing the function to modify the original value directly. This method is more efficient for large data structures but poses a risk of unintended modifications. 2. Explain recursion using functions in C with an example. (Handwritten)   #include <stdio.h>  *int* factorial(*int* *n*);  *int* main() {  *int* num;      printf("Enter a number: ");      scanf("%d", &num);      printf("Factorial of %d is %d\n", num, factorial(num));      return 0;  }  *int* factorial(*int* *n*) {      if (*n* == 0)          return 1;      return *n* \* factorial(*n* - 1);  } |

|  |
| --- |
| **Conclusion:** |
| We learned about User-Defined Functions (UDFs) and their significance in programming. Functions help modularize code, making it more structured, readable, and reusable. Instead of writing the same code repeatedly, we can define a function once and call it multiple times whenever needed, improving efficiency and maintainability. In this session, we specifically worked with function declaration, definition, and calling while also exploring parameter passing in functions. By passing parameters, we allow functions to work with different inputs dynamically rather than being restricted to fixed values. This enhances flexibility and reduces redundancy in the code. |

|  |
| --- |
| **Signature of faculty in-charge with Date:** |