Course Name:	**Programming in C**	**Semester:**	**II**
Date of Performance:	**17/02/2025**	**DIV/ Batch No:**	[redacted]
Student Name:	**MAYURI MANOJ KUMBHAR**	**Roll No:**	[redacted]

Experiment No: 5
Title: Strings and string handling functions

Aim and Objective of the Experiment:

Write a program in C to demonstrate use of strings and string handling functions.

COs to be achieved:

CO3: Apply the concepts of arrays and strings.

Theory:

In C programming, a string is an array of characters terminated by a null character ('\0'). Strings are represented using character arrays. To handle strings effectively, C provides a set of built-in functions in the <string.h> library.

Key functions for string:

- strlen(): Returns the length of a string (excluding the null-terminator).
- strcpy(): Copies a string from source to destination.
- strncpy(): Copies up to n characters from source to destination.
- strcat(): Appends one string to the end of another.
- strncat(): Appends up to n characters from source to destination.
- strcmp(): Compares two strings lexicographically.
- strncmp(): Compares the first n characters of two strings.
- strchr(): Searches for the first occurrence of a character in a string.
- strrchr(): Searches for the last occurrence of a character in a string.
- strstr(): Searches for the first occurrence of a substring in a string.
- strtok(): Tokenizes a string into substrings based on delimiters.
- sprintf(): Formats and stores a string into a character array.
- sscanf(): Reads formatted input from a string and stores it in variables.
- strdup(): Duplicates a string by allocating memory and copying it.
- strspn(): Returns the length of the initial segment of a string containing only characters from a set.
- strcspn(): Returns the length of the initial segment of a string excluding characters from a set.
- strpbrk(): Searches for the first occurrence of any character from a set in a string.

- strtok_r(): A reentrant version of strtok() for thread-safe tokenization.
- memcpy(): Copies memory from source to destination.
- memset(): Sets a block of memory to a specified value.

Problem Statements:

1. Write a program that takes a string as input and counts the number of vowels and consonants in the string without using the inbuilt library function. Ignore spaces and punctuation.
2. Write a program to manage student records. The program will handle the following operations using the string functions provided:
   - Input the student's name and grade (two strings).
   - Display the length of both the student's name and grade.
   - Copy the student's name into a new string and display it.
   - Concatenate a fixed string (e.g., " - Excellent Student") to the student's name and display the result.
   - Compare two students' names lexicographically and display which student has the lexicographically greater name.
   - Search for a substring in the student's name (e.g., "John" in "Johnny") and display the position of the first occurrence.
   - Search for a character in the grade string (e.g., 'A') and display the position of the first occurrence.
   - Tokenize the student's grade if it contains multiple components (e.g., "A B C") and display each component.

Code :

Question 1.]

```
#include <stdio.h>
int main() {
    char input_string[100];
    int vowel_count = 0;
    int consonant_count = 0;
    int i = 0;
    printf("Enter a string: ");
    fgets(input_string, sizeof(input_string), stdin);
    while (input_string[i]) {
        char char_lower = input_string[i];
        if (char_lower >= 'A' && char_lower <= 'Z') {
            char_lower += 32;
        }
        if (char_lower >= 'a' && char_lower <= 'z') {

            if (char_lower == 'a' || char_lower == 'e' || char_lower == 'i' ||
                char_lower == 'o' || char_lower == 'u') {
                vowel_count++;
            } else {
                consonant_count++;
            }
        }
        i++;
    }
    printf("Number of vowels: %d\n", vowel_count);
    printf("Number of consonants: %d\n", consonant_count);

    return 0;
}
```

Question 2.]

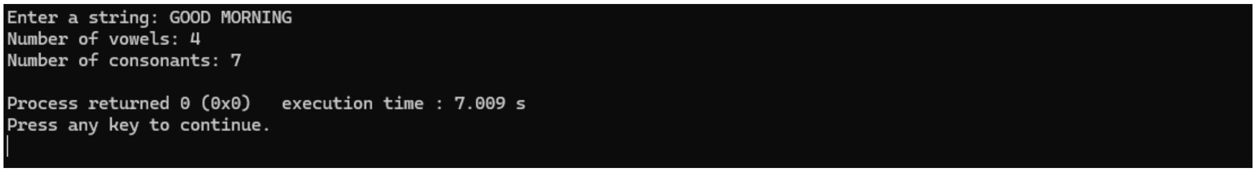
```
#include <stdio.h>
#include <string.h>
#define MAX_LEN 100
int main() {
    char name[MAX_LEN], grade[MAX_LEN];
    char copied_name[MAX_LEN];
    char search_substring[MAX_LEN];
    char search_char;
    char *pos;

    printf("Enter student's name: ");
    fgets(name, MAX_LEN, stdin);
    name[strcspn(name, "\n")] = '\0';

    printf("Enter student's grade: ");
    fgets(grade, MAX_LEN, stdin);
```

```
  grade[strcspn(grade, "\n")] = '\0';
  printf("\nLength of student's name: %lu", strlen(name));
  printf("\nLength of student's grade: %lu\n", strlen(grade));

  strcpy(copied_name, name);
  printf("\nCopied name: %s\n", copied_name);


  char full_name[MAX_LEN];
  strcpy(full_name, name);
  strcat(full_name, " - Excellent Student");
  printf("\concatenated string: %s\n", full_name);


  char name2[MAX_LEN];
  printf("\nEnter another student's name for comparison: ");
  fgets(name2, MAX_LEN, stdin);
  name2[strcspn(name2, "\n")] = '\0';


  int cmp_result = strcmp(name, name2);
  if (cmp_result > 0) {
     printf("\n%s is lexicographically greater than %s\n", name, name2);
  } else if (cmp_result < 0) {
     printf("\n%s is lexicographically smaller than %s\n", name, name2);
  } else {
     printf("\nBoth names are equal.\n");
  }

  printf("\nEnter a substring to search in the name: ");
  fgets(search_substring, MAX_LEN, stdin);
  search_substring[strcspn(search_substring, "\n")] = '\0';
  pos = strstr(name, search_substring);

  if (pos != NULL) {
     printf("\nSubstring found at position: %ld\n", pos - name);
  } else {
printf("\nSubstring not found in the name.\n");
  }
```

```
    printf("\nEnter a character to search in the grade: ");
    scanf(" %c", &search_char);

    pos = strchr(grade, search_char);
    if (pos != NULL) {
        printf("\nCharacter '%c' found at position: %ld\n", search_char, pos - grade);
    } else {
        printf("\nCharacter '%c' not found in the grade.\n", search_char);
    }

    printf("\nTokenized grade components:\n");
    char *token = strtok(grade, " ");
    while (token != NULL) {
        printf("%s\n", token);
        token = strtok(NULL, " ");
    }

    return 0;
}
```

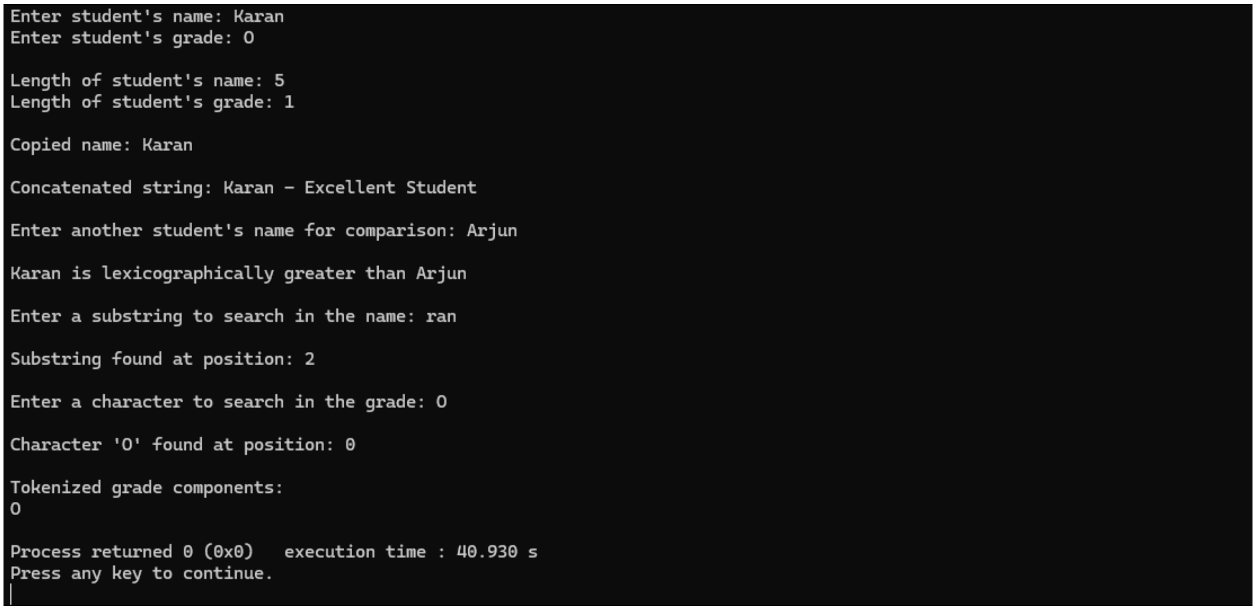
Output:

Question 1.]



```
Enter a string: GOOD MORNING
Number of vowels: 4
Number of consonants: 7

Process returned 0 (0x0)   execution time : 7.009 s
Press any key to continue.
```

Question 2.]



```
Enter student's name: Karan
Enter student's grade: O

Length of student's name: 5
Length of student's grade: 1

Copied name: Karan

Concatenated string: Karan - Excellent Student

Enter another student's name for comparison: Arjun

Karan is lexicographically greater than Arjun

Enter a substring to search in the name: ran

Substring found at position: 2

Enter a character to search in the grade: O

Character 'O' found at position: 0

Tokenized grade components:
O

Process returned 0 (0x0)   execution time : 40.930 s
Press any key to continue.
```

Post Lab Subjective/Objective type Questions:

1. In C, what will happen if you pass an uninitialized string or a string without a null terminator to any of the string handling functions (e.g., strcpy(), strlen(), strcmp())?

Ans: *Passing an uninitialized string or a string without a null terminator to any of the string handling functions in C can lead to undefined behavior. This is because these functions rely on the presence of the null terminator (\0) to determine the end of the string.*
Here are some possible consequences:

- *strcpy(): If you use strcpy() to copy a string without a null terminator, it may result in copying random memory contents until it encounters a null terminator, potentially causing a buffer overflow and overwriting adjacent memory.*
- *strlen(): Calling strlen() on an uninitialized string or a string without a null terminator will result in reading beyond the intended memory, as it will keep counting until it finds a null terminator. This can cause the program to crash or exhibit unpredictable behavior.*
- *strcmp(): Using strcmp() with such strings will lead to comparing memory contents beyond the intended strings. This can produce incorrect results, access violations, or crashes.*

2. In C, how does memory allocation for strings work? What are the potential risks associated with string manipulation in C, and how can buffer overflow issues be prevented?

Ans: *There are two types of string allocation in C :*

- *Static Allocation: Declaring a fixed-size array.*
- *Dynamic Allocation: Using functions like malloc() or calloc() to allocate memory at runtime.*

Potential Risks with String Manipulation

- *Buffer Overflows: Writing more data than the allocated memory can handle, which can overwrite adjacent memory.*
- *Uninitialized Strings: Using strings that have not been properly initialized can lead to unpredictable behavior.*
- *String Termination: Forgetting to null-terminate a string can cause functions to read beyond the intended memory.*

Preventing Buffer Overflow Issues

- *Bounds Checking: Always check the length of the string and ensure it fits within the allocated memory.*
- *Proper Memory Allocation: Allocate sufficient memory to accommodate the string and the null terminator.*
- *Use Safe Functions: Prefer safer functions like strncpy() over strcpy(), snprintf() over sprintf(), etc.*

Conclusion:

In this experiment, we learnt about strings, string handling functions and concepts of arrays.
A string is an array of characters terminated by a null character ('\0'). Strings are represented using character arrays. To handle strings effectively, C provides a set of built-in functions in the <string.h> library.

Signature of faculty in-charge with Date: