| | | | |
|---|---|---|---|
| **Course Name:** | **Programming in C** | **Semester:** | **II** |
| **Date of Performance:** | **12/03/25** | **DIV/ Batch No:** | **C2-2** |
| **Student Name:** | **Ashwera Hasan** | **Roll No:** | **16010124107** |

# 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 :**

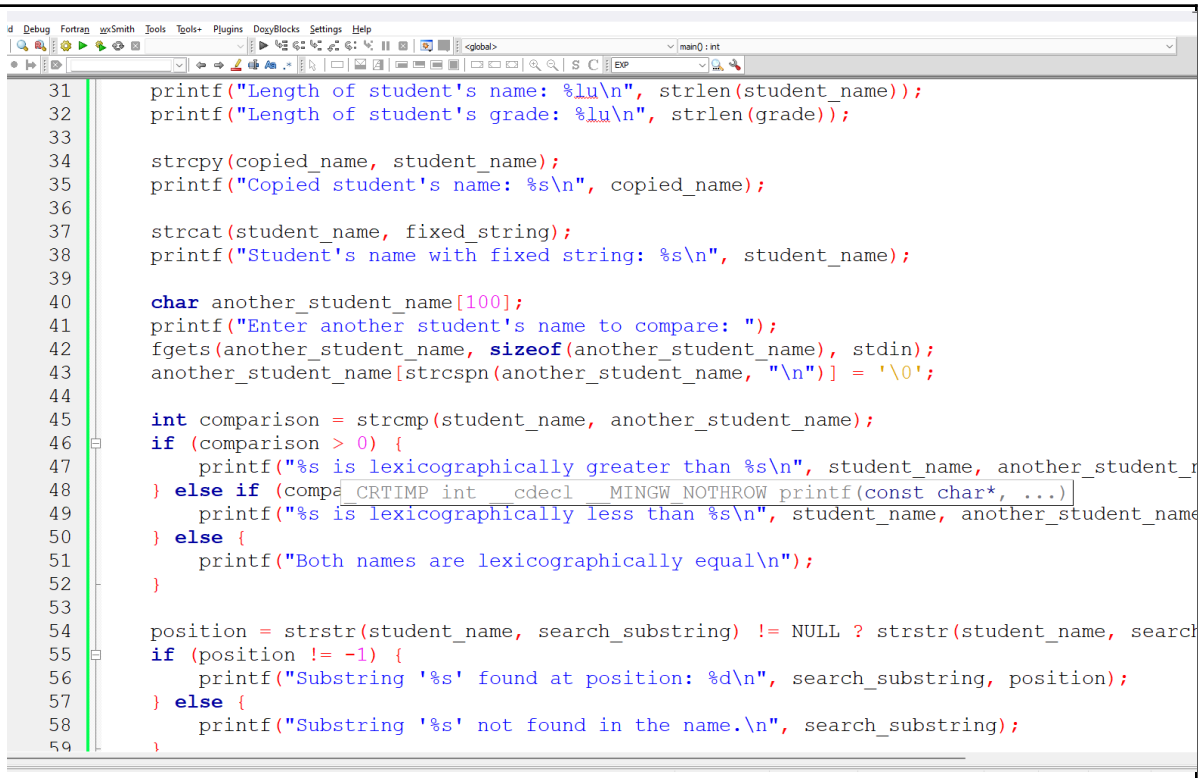
1.

```c
#include <stdio.h>
#include <ctype.h>


void count_vowels_consonants(const char *str, int *vowels, int *consonants) {
    *vowels = 0;
    *consonants = 0;

    while (*str != '\0') {
        char ch = *str;

        if (isalpha(ch)) {
            if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u'
                (*vowels)++;
            } else {
                (*consonants)++;
            }
        }
        str++;
    }
}
```

```c

int main() {
    char str[100];
    int vowels = 0, consonants = 0;

    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

    count_vowels_consonants(str, &vowels, &consonants)

    printf("Number of vowels: %d\n", vowels);
    printf("Number of consonants: %d\n", consonants);

    return 0;
}
```

2.

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

void tokenize_grade(char *grade) {
    char *token = strtok(grade, " ");

    while (token != NULL) {
        printf("Grade component: %s\n", token);
        token = strtok(NULL, " ");
    }
}

int main() {
    char student_name[100], grade[100];
    char copied_name[100];
    char search_substring[] = "John";
    char search_char = 'A';
    char fixed_string[] = " - Excellent Student";
    int position;

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

    printf("Enter the student's grade: ");
    fgets(grade, sizeof(grade), stdin);
    grade[strcspn(grade, "\n")] = '\0';
```



```
    printf("Length of student's name: %lu\n", strlen(student_name));
    printf("Length of student's grade: %lu\n", strlen(grade));

    strcpy(copied_name, student_name);
    printf("Copied student's name: %s\n", copied_name);

    strcat(student_name, fixed_string);
    printf("Student's name with fixed string: %s\n", student_name);

    char another_student_name[100];
    printf("Enter another student's name to compare: ");
    fgets(another_student_name, sizeof(another_student_name), stdin);
    another_student_name[strcspn(another_student_name, "\n")] = '\0';

    int comparison = strcmp(student_name, another_student_name);
    if (comparison > 0) {
        printf("%s is lexicographically greater than %s\n", student_name, another_student_r
    } else if (compa__CRTIMP int __cdecl __MINGW_NOTHROW printf(const char*, ...)
        printf("%s is lexicographically less than %s\n", student_name, another_student_name
    } else {
        printf("Both names are lexicographically equal\n");
    }

    position = strstr(student_name, search_substring) != NULL ? strstr(student_name, search
    if (position != -1) {
        printf("Substring '%s' found at position: %d\n", search_substring, position);
    } else {
        printf("Substring '%s' not found in the name.\n", search_substring);
    }
```

```
    printf("Both names are lexicographically equal\n");
}

position = strstr(student_name, search_substring) != NULL ? strstr(student_name, sear
if (position != -1) {
    printf("Substring '%s' found at position: %d\n", search_substring, position);
} else {
    printf("Substring '%s' not found in the name.\n", search_substring);
}


position = strchr(grade, search_char) != NULL ? strchr(grade, search_char) - grade :
if (position != -1) {
    printf("Character '%c' found at position: %d\n", search_char, position);
} else {
    printf("Character '%c' not found in the grade.\n", search_char);
}

return 0;
```

**Output:**

1.
```
"D:\C1-2 39\try.exe"
Enter a string: ashwera hasan is a rockstar
Number of vowels: 9
Number of consonants: 14

Process returned 0 (0x0)   execution time : 47.628 s
Press any key to continue.
```

2.
```
"D:\C1-2 39\try.exe"
Enter the student's name: Ashwera Hasan
Enter the student's grade: 98.6
Length of student's name: 13
Length of student's grade: 4
Copied student's name: Ashwera Hasan
Student's name with fixed string: Ashwera Hasan - Excellent Student
Enter another student's name to compare: Shruti Bhaskar
Ashwera Hasan - Excellent Student is lexicographically less than Shruti Bhaskar
Substring 'John' not found in the name.
Character 'A' not found in the grade.

Process returned 0 (0x0)   execution time : 77.616 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())?

Passing an uninitialized string or one without a null terminator to functions like strcpy(), strlen(), or strcmp() results in undefined behavior, including segmentation faults, memory corruption, or incorrect results. These functions rely on a null terminator to determine string boundaries, and its absence can cause unpredictable issues. Compiler does not understand where to stop string functionality and results in error.

**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?

In C, strings are arrays of characters terminated by \0. Memory for strings can be allocated statically (e.g., char str[100];) or dynamically (e.g., char *str = malloc(100);). Risks include

buffer overflow, where a string exceeds allocated memory, overwriting adjacent data, and causing crashes or vulnerabilities.

To prevent buffer overflows, we should use safer functions like strncpy(), snprintf(), or strncat(), and always ensure strings are null-terminated. Additionally, bounds checking with sizeof() and validating string lengths before manipulation helps mitigate these risks.

**Conclusion:**

String functionality in C allows us to work with strings, including tasks like determining their length, extracting substrings, and concatenation. However, since C is pointer-based, it's crucial to ensure strings are null-terminated (\0). This enables the compiler to properly identify the string's end and prevents undefined behavior caused by missing null terminators.

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