Session 7: Strings in C

1. Title + Big Idea (The Setup)

Topic Name: Strings in C

One-line purpose: To understand how to represent and manipulate sequences of characters (text) in C programming using character arrays.

2. Why Do We Need It? (The Problem)

Problem it solves: Programs often need to handle textual data, such as names, messages, or file paths. C doesn't have a built-in `String` type like some other languages, so character arrays are used to store and process strings. Without a standardized way to handle text, programs would be limited to numerical operations.

Real-life analogy: A string in C is like a word written on a series of individual cards, where each card holds one letter. To know where the word ends, you put a special blank card (the null terminator) at the very end. This allows you to read the whole word character by character until you hit the blank card.

3. Core Concept / Definition (The Truth)

Simple definition: In C, a **string** is an array of characters terminated by a null character (`\0`). This null character signifies the end of the string, allowing functions to correctly process its length and content. Strings are fundamental for handling any form of text data.

Key rule(s):

- Strings are essentially **character arrays**.
- Every string **must end with a null terminator (** `\0` **)**.
- String manipulation often involves functions from the `<string.h>` library.

- When declaring a character array for a string, ensure it has enough space for all characters **plus one extra byte for the null terminator**.

4. Visual / Flow / Diagram (Show, Don't Tell)

String Representation in Memory:

```
String: "Hello"

Memory Address:  1000   1001   1002   1003   1004   1005
                +------+------+------+------+------+------+
Characters:     |  H   |  e   |  l   |  l   |  o   | \0   |
                +------+------+------+------+------+------+
Indices:          [0]    [1]    [2]    [3]    [4]    [5]
```

5. Syntax / Structure (The Tool)

Declaration and Initialization:

- **Using character array:**

```
char str1[20];             // Declares a character array of size 20
char str2[] = "Hello";    // Initializes with a string literal, size
is 6 (5 chars + \0)
char str3[10] = {"World"}; // Initializes with a string literal
char str4[6] = {"H", "e", "l", "l", "o", "\0"}; // Explicit character
initialization
```

Input/Output:

- **Input using `scanf()`:**

```
char name[50];
printf("Enter your name: ");
scanf("%s", name); // Reads a single word, stops at whitespace
```

- **Input using `fgets()` (safer for multi-word strings):**

  ```c
  char sentence[100];
  printf("Enter a sentence: ");
  fgets(sentence, sizeof(sentence), stdin); // Reads a line including spaces
  ```

- **Output using `printf()`:**

  ```c
  char greeting[] = "Hello, C!";
  printf("%s\n", greeting); // Prints the entire string until \0
  ```

Common String Functions (from `<string.h>`):

Function	Description
`strlen(str)`	Returns the length of the string (excluding `\0`)
`strcpy(dest, src)`	Copies `src` string to `dest` string
`strcat(dest, src)`	Concatenates `src` string to `dest` string
`strcmp(str1, str2)`	Compares `str1` and `str2` lexicographically
`strchr(str, char)`	Finds the first occurrence of a character
`strstr(str, sub)`	Finds the first occurrence of a substring

6. Example (The Action)

```c
#include <stdio.h>
#include <string.h> // Required for string functions

int main() {
    char firstName[20];
    char lastName[20];
    char fullName[40];

    // Input first name
    printf("Enter your first name: ");
    scanf("%s", firstName); // scanf stops at whitespace

    // Input last name
    printf("Enter your last name: ");
    scanf("%s", lastName);

    // Get length of first name
    printf("Length of first name: %zu\n", strlen(firstName)); // %zu for size_t

    // Copy first name to full name
    strcpy(fullName, firstName);

    // Concatenate last name to full name
    strcat(fullName, " "); // Add a space
    strcat(fullName, lastName);

    // Print full name
    printf("Your full name is: %s\n", fullName);

    // Compare strings
    if (strcmp(firstName, "John") == 0) {
        printf("Hello, John!\n");
    } else {
        printf("You are not John.\n");
    }

    return 0;
}
```

7. Common Mistakes (The Conflict)

1. **Buffer Overflow:** Not allocating enough space for the character array, especially when using `strcpy()` or `strcat()`, can lead to writing beyond the allocated memory, causing crashes or security vulnerabilities.
2. **Forgetting Null Terminator:** Manually manipulating character arrays without ensuring the `\0` at the end can lead to functions reading past the intended end of the string, resulting in garbage output or crashes.
3. **Using `=` for String Assignment:** Attempting to assign one string to another using the `=` operator (e.g., `str1 = str2;`) will not work for character arrays; `strcpy()` must be used instead.
4. **`scanf("%s", ...)` with Spaces:** `scanf("%s", ...)` reads only a single word and stops at the first whitespace character. For reading entire lines with spaces, `fgets()` is preferred.

8. Key Points / Rules (The Takeaway)

- Strings in C are null-terminated character arrays.
- The null terminator (`\0`) is crucial for string functions to identify the end of the string.
- Always allocate sufficient memory for strings, including space for the null terminator.
- Use functions from `<string.h>` (e.g., `strcpy`, `strcat`, `strlen`, `strcmp`) for string manipulation.
- Be cautious with `scanf("%s", ...)` for input; `fgets()` is generally safer for multi-word input.

9. One-Line Summary (The Ending)

Strings in C are managed as null-terminated character arrays, providing a flexible yet careful approach to handling and manipulating textual data through dedicated library functions.

10. (Optional) Practice Trigger

Question: Write a C program that takes a string as input from the user and then prints the string in reverse order.

Variation Idea: Create a program that counts the number of vowels and consonants in a given string.