Department of Computer Science and Engineering,
PES University, Bangalore, India

Lecture Notes
Problem Solving With C
UE24CS151B

Lecture #4
Errors and Best Practices associated with String Manipulation Functions

By,
Prof. Sindhu R Pai,
Theory Anchor, Feb-May, 2025
Assistant Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Center, PES University)
Prof. Nitin V Poojari (Dean, Internal Quality Assurance Cell, PES University)

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Errors and best Practices associated with String Manipulation Functions

Course objectives: The objective(s) of this course is to make students

- Acquire knowledge on how to solve relevant and logical problems using computing Machine.
- Map algorithmic solutions to relevant features of C programming language constructs.
- Gain knowledge about C constructs and its associated ecosystem.
- Appreciate and gain knowledge about the issues with C Standards and it's respective behaviours.

Course outcomes: At the end of the course, the student will be able to:

- Understand and Apply algorithmic solutions to counting problems using appropriate C Constructs.
- Understand, Analyze and Apply sorting and Searching techniques.
- Understand, Analyze and Apply text processing and string manipulation methods using Arrays, Pointers and functions.
- Understand user defined type creation and implement the same using C structures, unions and other ways by reading and storing the data in secondary systems which are portable.

Sindhu R Pai

Theory Anchor, Feb - May, 2025

Dept. of CSE,

PES University

Introduction

String manipulation functions, while powerful and efficient, can lead to serious issues if not used carefully. This lecture notes highlights common errors developers encounter—such as **buffer overflows, improper null termination, and memory corruption**—and provides insights into how these problems arise and **how they can be effectively avoided through best practices**.

Buffer Overflow:

A buffer is **a temporary, fixed-size block of memory** used to store data—such as characters in a string while it is being transferred or processed. In programming, buffers help manage data efficiently, especially during input/output operations or string manipulations. **Buffer overflow occurs when writing beyond the bounds of the destination buffer.**

Coding Example_1:

```
char dest[5];
strcpy(dest, "Hello, World!");  // Overflows dest buffer
```

The above problem can be avoided by ensuring that the **destination buffer is large enough and use safer alternatives like strncpy.**

```
char dest[20];
strncpy(dest, "Hello, World!", sizeof(dest) - 1);
dest[sizeof(dest) - 1] = '\0';  // Ensure null termination
```

Missing Null Terminator:

Some functions do not automatically null-terminate strings if the size limit is reached.

Coding Example_2:

```
char dest[5];
strncpy(dest, "Hello", 5);  // Copies 5 characters, but no space for '\0'
printf("%s\n", dest);       // May cause undefined behavior
```

The above problem can be avoided by ensuring that the **destination buffer is large enough and use safer alternatives like strncpy.**

```
char dest[6];  // observe this
snprintf(dest, sizeof(dest), "%s", "Hello");
// snprintf automatically ensures null termination as long as the buffer size is respected.
printf("%s\n", dest);
```

Coding Example_3: Using strlen on a string that is not null-terminated.

```
char arr[5] = {'H', 'e', 'l', 'l', 'o'};  // No '\0'
printf("%zu", strlen(arr));  // Unsafe
```

The above problem can be avoided by ensuring that the **data is within double quotes and memory is large enough to hold the data.**

```
char arr[6] = "Hello";  // Automatically null-terminated
printf("%zu", strlen(arr));  // 5
```

Using Uninitialized Pointers:

An uninitialized pointer is a pointer variable that has been declared but not assigned any valid memory address before being used. **Performing string operations on uninitialized pointers can lead to segmentation faults.**

Coding Example_4:

```
char *str;            // Uninitialized pointer
strcpy(str, "Hello");
// str doesn't point to valid memory. So copying a string into it is dangerous
```

The above problem can be avoided **by initializing the pointers before use—either by assigning them to a valid variable's address or by dynamically allocating memory.**

```
char *str = malloc(20);      // Allocate memory
strcpy(str, "Hello");        // Safe
printf("%s\n", str);
free(str);                   // Best practice related to DMA functions
```

When working with string manipulation functions in C, even small mistakes can lead to serious issues such as **crashes, memory corruption, or security vulnerabilities**. To ensure safe and reliable code, it's essential to **follow certain best practices**. The following tips help prevent common pitfalls and promote robust string handling.

1. **Allocate Sufficient Buffer Size**
   - Always include space for the **null terminator** (\0) when defining string buffers.
2. **Use Safer Alternatives**
   - Prefer strncpy, strncat, and snprintf over strcpy, strcat, and sprintf.
3. **Manually Add Null Terminator if Needed**
   - Functions like strncpy may not null-terminate—do it manually.
4. **Check Buffer Sizes Before Copying**
   - Prevent buffer overflows by validating destination size before any copy/append.
5. **Initialize Pointers Properly**
   - Don’t use pointers before assigning them a valid memory address.
6. **Free Allocated Memory**
   - Use free() to avoid memory leaks when memory is allocated with malloc.
7. **Validate Strings Before Using strlen**
   - Ensure the string is null-terminated before calling strlen or similar functions.
8. **Sanitize and Validate Input**
   - Always check the size and validity of input strings, especially from user input.

To further strengthen the safety of your code, avoid using functions like strcpy or memcpy when the source and destination memory areas may overlap—opt for **memmove, which is designed to handle such cases safely**. Additionally, make use of compiler warnings (such as -Wall and -Wextra) and static analysis tools to catch potential issues early in the development cycle. These proactive measures help in identifying risky patterns and maintaining the overall reliability and security of your string operations.

Avoid hidden bugs and ensure smooth execution of string manipulation functions!!!