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

Lecture Notes
Problem Solving With C
UE24CS151B

Lecture #3
String Manipulation Functions in C

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: String Manipulation Functions in C

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

We know that in C, **strings** are represented as arrays of characters ending with a **null character ('\0')**. Since C **does not have a built-in string type** like some other languages such as Python and Javascript, string manipulation is done using **standard library functions** provided in the **string.h** header file. These functions help perform **common operations** like:

- Copying strings
- Concatenating strings
- Comparing strings
- Finding the length of a string
- Searching within strings

The <string.h> provides **around 25 standard string and memory functions**, although the **exact count may vary slightly** depending on the compiler and platform. The below table lists **most commonly used string related functions in from string.h**, including:

- **What each function does**
- **What it returns on success**
- **What it returns on failure (if applicable)**

Function Name	Description	Return values
`strlen(str)`	Returns the number of characters in `str` (excluding `\0`)	Length of the string (`size_t`) on success; undefined behavior if `str` is `NULL`
`strcpy(dest, src)`	Copies `src` into `dest` including the null terminator	Pointer to `dest` on success; undefined behavior if `src` or `dest` is `NULL` or overlap
`strncpy(dest, src, n)`	Copies up to `n` characters from `src` to `dest`	Pointer to `dest`; may not null-terminate if `src` is longer than `n`; undefined if `NULL`
`strcat(dest, src)`	Appends `src` to the end of `dest`	Pointer to `dest` on success; undefined behavior if strings overlap or `dest` is too small
`strncat(dest, src, n)`	Appends up to `n` characters of `src` to `dest`	Pointer to `dest`; undefined behavior if space is insufficient
`strcmp(str1, str2)`	Compares `str1` and `str2`	0 if equal, `<0` if `str1 < str2`, `>0` if `str1 > str2`; undefined if `NULL`
`strncmp(str1, str2, n)`	Compares up to `n` characters of two strings	Same as `strcmp`; undefined if either string is `NULL`
`strchr(str, ch)`	Finds first occurrence of `ch` in `str`	Pointer to first match on success; `NULL` if not found
`strrchr(str, ch)`	Finds last occurrence of `ch` in `str`	Pointer to last match on success; `NULL` if not found
`strstr(haystack, needle)`	Finds first occurrence of `needle` in `haystack`	Pointer to match on success; `NULL` if not found
`strtok(str, delim)`	Splits `str` into tokens using `delim`	Pointer to next token on success; `NULL` if no more tokens

Note:

- Most functions **return a pointer** to allow chaining and efficiency.
- Many functions exhibit **undefined behavior if passed NULL pointers** or **non-null-terminated strings**.
- strtok() maintains internal state, so it is **not thread-safe** — use strtok_r() in multi-threaded programs.

Coding Example_1: Demo of all above functions with simple examples

```
#include <stdio.h>
#include <string.h>
int main() {
   char str1[100] = "Hello";
   char str2[] = "World";
   char str3[100];
   char str4[100];

   // strlen
   printf("Length of str1: %lu\n", strlen(str1));  // Output: 5

   // strcpy       // Compiletime Error
   str3 = str1;  // we are trying to equate two addresses. Array Assignment incompatible
   strcpy(str3, str1);
   printf("After strcpy, str3: %s\n", str3);

   // strncpy
   strncpy(str4, str2, 3);
   str4[3] = '\0';  // manually null-terminate
   printf("After strncpy, str4: %s\n", str4);

   // strcat
   strcat(str1, " ");     strcat(str1, str2);
   printf("After strcat, str1: %s\n", str1);  // Make sure str1 is big enough to hold str1 and str2 both.
```

```
Length of str1: 5
After strcpy, str3: Hello
After strncpy, str4: Wor
After strcat, str1: Hello World
After strncat, str3: HelloWor
```

```
// strncat
strncat(str3, str2, 3);
printf("After strncat, str3: %s\n", str3);  // Output: HelloWor

// strcmp
printf("strcmp(\"abc\", \"abd\"): %d\n", strcmp("abc", "abd"));  // < 0

// strncmp
printf("strncmp(\"abc\", \"abd\", 2): %d\n", strncmp("abc", "abd", 2));  // 0

// strchr
char *ptr = strchr(str1, 'W');
if (ptr) {
    printf("First 'W' in str1 at index: %ld\n", ptr - str1);  // Output: 6
}

// strrchr
char testStr[ ] = "banana";
char *last_a = strrchr(testStr, 'a');
if (last_a) {
    printf("Last 'a' in %s at index: %ld\n", testStr, last_a - testStr);  // Output: 5
}

// strstr
char *found = strstr(str1, "World");
if (found) {
    printf("\"World\" found at position: %ld\n", found - str1);  // Output: 6
}
```

```
strcmp("abc", "abd"): -1
strncmp("abc", "abd", 2): 0
First 'W' in str1 at index: 6
Last 'a' in banana at index: 5
"World" found at position: 6
```

```
    // strtok
    char fruits[ ] = "apple,banana,grape";
    char *token = strtok(fruits, ",");
    while (token != NULL) {
        printf("Token: %s\n", token);
        token = strtok(NULL, ",");
    }
    return 0;
}
```

```
Token: apple
Token: banana
Token: grape
```

Now let us try writing the user defined functions to understand few operations in detail.

Coding Example_2: User defined function to find the string length. Client code is given.

```
char mystr[ ] = "pes";
printf("Length is %d\n", my_strlen(mystr));
```

Different versions of my_strlen( ) implementation are as below.

Version 1: Iterate through every character of the string using a variable. Stop iteration when NULL character is encountered and return the value of that variable.

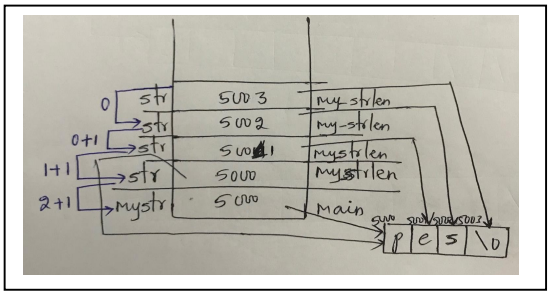
```
int my_strlen(char str[])
{       int i = 0;
        while(str[i] != '\0')    {       ++i;    }
        return i;
 }
```

Version 2: Run the loop till *s becomes '\0' character. Increment the pointer and the counter when *str is not NULL.

```
int my_strlen(char *str)
{       int i =0;
        while(*str){                    // str[i] != '\0' // *(str+i) != '\0' ---> all these are same
                i++;    str++;    }
        return i;
}
```

Version 3: Using recursion and pre-increment

```
int my_strlen(char *str)
{       if (!(*str))       return 0;
          else             return 1+my_strlen(++str);
}
```

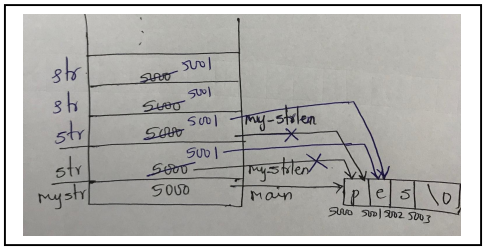


Version 4: Using recursion and no change in the pointer in the function call

```
int my_strlen(char *str)
{       if (!(*str))       return 0;
        else return 1+my_strlen(str+1);
}
```

Version 5: Using recursion and post-increment

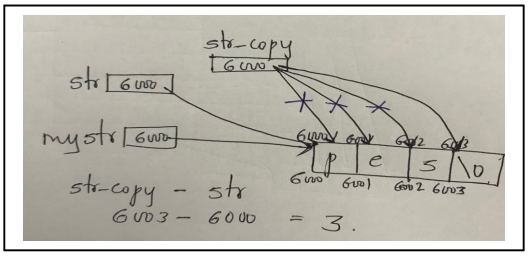
```
int my_strlen(char *str)
{       if (!(*str)) return 0;
        else   return 1+my_strlen(str++);
}
```



Version 6: Using Pointer arithmetic

Use a local pointer which points to the first character of the string. Keep incrementing this till '\0' is found. Then subtract this from the pointer specified in the parameter which points to the beginning of the string. This finds the length of the string.

```
int my_strlen(char *str)
{
        char *str_copy = str;
        while(str_copy){     str_copy++;     }
        return str_copy - str;
}
```



Coding Example_3: User defined function to copy the given string. Client code is given.

```
char mystr1[ ] = "pes university";
char mystr2[100];
printf("%s\n",mystr1);
my_strcpy(mystr2, mystr1);
printf("%s\n",mystr2);
```

Version 1: Implementation of my_strcpy( ) by passing arrays to function and using array notation

```
void my_strcpy(char *b, char *a)
{
        // copy a to b
        int i =0;
        while(a[i] != '\0')
        {
                b[i]=a[i];
                i++;
        }
        b[i] = '\0';        // append '\0' at the end
  }
```

Version 2: Implementation of my_strcpy( ) by passing arrays to function and using pointer notation

```
void my_strcpy(char *b, char *a)
{       // copy a to b
        while(*a != '\0')
        {    *b = *a; b++; a++;       }
         *b = '\0';         // append '\0' at the end
}
```

Version 3: Implementation of my_strcpy( ) using pointer notation

```
void my_strcpy(char *b, char *a)
{
        while((*b = *a) != '\0')
        {               b++;   a++;     }
}
```

Version 4: Implementation of my_strcpy( ) using pointer notation

```
void my_strcpy(char *b, char *a)
{
        while(*b++ = *a++);          // same as for(;*b++ = *a++;);
}
```

Coding Example_4: User defined function to compare the given two strings. Client code is given.

```
char str1[100]; char str2[100];
printf("enter the first string\n"); scanf("%s",str1);
printf("enter the second string\n"); scanf("%s",str2);
int res = my_strcmp(str1,str2);
printf("result is %d\n",res);
if(!res)
        printf("%s and %s are equal\n",str1,str2);
else if(res > 0)
        printf("%s is higher than %s\n",str1,str2);
else
        printf("%s is lower than %s\n",str1,str2);
```

Version 1: my_strcmp( ) returns < 0 if a < b, 0 if a == b , > 0 if a > b

```
int my_strcmp(char *a, char *b)
{
        int i;
        for(i = 0; b[i]!= '\0' && a[i] != '\0' && b[i]==a[i]; i++);
        return a[i]-b[i];
}
```

Version 2: Using pointer notation

```
int my_strcmp(char *a, char *b)
{
        for(;*b && *a && *b == *a; a++,b++);
        return *a - *b;
}
```

Coding Example_5: User defined function to find the address and the position of character in a given string. Client code is given.

```
char str1[100];
printf("enter the string\n");
scanf("%s",str1);
printf("enter the character\n");
char ch = getchar();
char *p = my_strchr(str, ch);
printf("present in this address %d",p);
if(p)
        printf("present in %d position\n", p - a);
else
        printf("character not present\n");
```

Version 1:

```
char* mystrchr(char *a, char c)
{       char *p = NULL;
        char *s = a;
        while(*s != '\0' && p==NULL)
        {       if(*s == c)
                        p = s;
                s++;
        }
        return p;
}
```

Version 2:

```
char *mystrchr(char *a,char c)
{       while(*a && *a != c)
        {       a++;    }
        if (!(*a))  return NULL;
        else    return a;
        // Can replace if else with return !(*a)? NULL: a;
}
```

Coding Example_6: User defined function to concatenate the given two strings. Client code is given.

```
char str1[100];
char str2[100];
printf("enter first string\n"); scanf("%s",str1);
printf("enter second string\n"); scanf("%s",str2);
my_strcat(str1,str2);
printf("str1 is %s and str2 is %s\n",str1,str2);
```

Version 1:Using Array notation on the pointer

```
void my_strcat(char *a, char *b)
{       int i = 0;
        while(a[i]!= '\0')   {    i++;    }        // while can be replaced with strlen() int j;
        for(j = 0;b[j] != '\0';j++,i++)
                a[i] = b[j];
        a[i] = '\0';
}
```

Version 2:Using Pointer notation on the pointer

```
void my_strcat(char *a,char *b)
{       while(*a){ a++; };      // increment a till NULL.
         while(*b)      {        *a = *b;   a++;b++;    }
        // Once NULL, copy each character from b to a and increment a and b both till b becomes '\0'
        *a = '\0';         // assign '\0' character to a's end
}
```

While these functions are efficient, they require **careful attention to memory management, null terminators, and buffer sizes to prevent common pitfalls such as buffer overflows, memory corruption, and undefined behavior.** In this regard, typical mistakes that occur during string manipulation and best practices to avoid them are explored in the next Lecture notes.

Happy Coding using String Functions!!