IAE CLUB

By Oussama Laamri & Saad Larhrib

March 6, 2025

House of Code: Phase1 ”String Manipulation in C”

Contents

1 Introduction to the Challenge

Welcome to Phase1 – A House of Code Challenge!
This Phase(challenge) is all about sharpening your problem-solving skills through practical coding exercises. It's designed for anyone who wants to practice and get better at tackling coding problems, no matter your experience level.

What Is This Challenge About?

In this challenge, you will solve problems based on string manipulation and basic algorithms. The goal isn't necessarily to win but to improve your problem-solving abilities and learn from your mistakes and successes.

Why C Language?

For this challenge, you are required to use the C programming language. This choice has been made because all of you are already familiar with C, and it's a language that is widely used in programming challenges, particularly for algorithms and low-level system programming. By using C, we can easily evaluate how well your code respects basic programming rules such as memory management and efficiency. Additionally, since you are already comfortable with C, it will allow us to focus more on problem-solving and algorithm design, rather than on learning new programming concepts.

Challenge Goals:

- **Learn & Improve:** The main purpose of this challenge is to practice coding, understand problem-solving techniques, and get better with each phase.
- **It's Okay If You Don't Win:** Don't worry if you don't solve all the problems or get everything right. The challenge is for learning, and everyone progresses at their own pace.
- **Try Your Best:** Even if you don't know something, Google is your friend! Use the resources we provide, search online, or ask the community for help.
- **The Goal Is Progress:** Whether you solve 1% or 100% of the problems, as long as you learn something new, you're progressing!

How We Will Grade Your Code:

We will grade your submissions based on the following criteria:

- **Effort (1 Point):** You will receive 1 point just for attempting to solve the problem, even if your code doesn't pass all the tests. *It's important that you try!*
- **Correctness (1-4 Points):** Points will be awarded if your code passes all the provided test cases and is correct. The more tests your code passes, the higher the score.

- **Code Quality (1-2 Points):** Clean and well-structured code will earn extra points. We look for clarity in your logic, appropriate use of functions, and good variable names.
- **Comments (1 Point):** If your code includes helpful comments explaining your logic and steps, you will receive extra points. This makes your code more understandable and easier to follow.

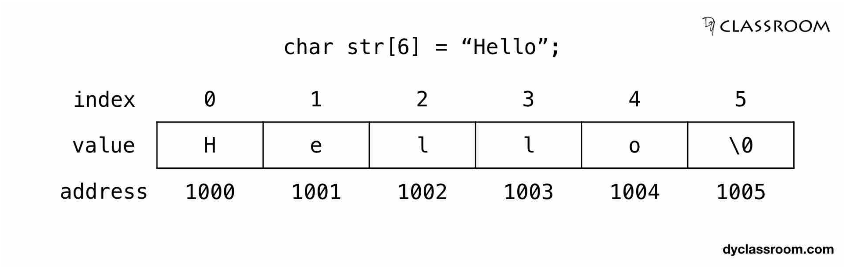
Important Notes:

- **Don't Use AI (ChatGPT, Deep Seek, etc.):** While learning resources and Google are allowed, using AI to solve problems for you is **not allowed**. Solve problems on your own to improve your skills.
- **Collaboration Is Fine:** It's great to discuss problems with others if you don't understand something. Just be careful not to share full solutions. Instead, help each other learn by explaining ideas or concepts.
- **Focus on Learning:** The main goal of this challenge is to learn and improve problem-solving skills, not to win. Don't stress if you don't solve everything on time. Just keep moving forward and learning.

2 Introduction to Strings in C

In C programming, a string is essentially a sequence of characters. Unlike some other programming languages where strings are special data types, in C, strings are simply arrays of characters. Each character in a string is stored in a contiguous memory location, one after the other, just like elements in an array.
The most important thing to remember about strings in C is that they are **terminated** with a special character called the **null terminator**. This null terminator is represented by the symbol `\0` and indicates the end of the string.

2.1 Example



This array of characters includes the characters of the word "Hello" followed by the null character `\0`, which signifies the end of the string.

2.2 Why is the Null Terminator Important?

The null terminator is vital because it helps C functions know where the string ends. Without it, functions like string concatenation, copying, and comparison wouldn't work correctly because there would be no way to tell where the string ends.
For example, if we try to print the string without the null terminator, we would likely print unwanted memory content after the intended string, leading to unexpected behavior. Here's an example:

```
char str[] = {'H', 'e', 'l', 'l', 'o'};
printf("%s", str); // This will not work correctly
```

This will lead to undefined behavior, as there's no `\0` to mark the end of the string. It's important to always make sure that strings in C are properly terminated.

2.3 String Operations in C

The null terminator also plays a key role when performing operations on strings. Common operations include:

- **Concatenation**: Joining two strings together.
- **Copying**: Duplicating one string into another.
- **Comparison**: Checking if two strings are equal or determining their lexicographical order.

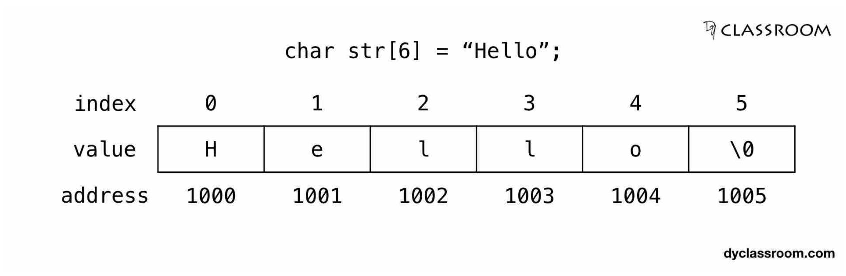
These operations rely on the presence of the `\0` character to understand where the string ends.
For example, consider the following code for string concatenation:

```
char str1[20] = ”Hello”;
char str2[] = ”World”;
strcat(str1, str2);
printf(”%s”, str1);  // Output: ”HelloWorld”
```

The `strcat` function appends the string `str2` to `str1` by locating the null terminator of `str1` and then appending `str2` starting from there. Without the null terminator in `str1`, `strcat` wouldn't know where to begin appending.

2.4 Conclusion

In C, strings are just arrays of characters, and the null terminator (`\0`) marks the end of the string. This simple yet important detail makes string operations in C both powerful and tricky. Always ensure that strings are properly terminated with `\0` to avoid errors and unexpected behaviors in your programs.



```
char str[6] = “Hello”;
```

index	0	1	2	3	4	5
value	H	e	l	l	o	\0
address	1000	1001	1002	1003	1004	1005

3 Resources

- Arrays in C (Article)
- Pointers in C (Article)
- Strings in C (Article)
- String Basics (video)
- String In Char Array VS. Pointer To String Literal (Video)
- Stack Data Structure (Video)
- Search on Google if you still don't understand something about strings in C.

4 General Rules

- You are only allowed to use the standard libraries `<stdio.h>` and `<stdbool.h>`. You are not allowed to use any other libraries.
- You may write helper functions if needed, and submit them with your main code.
- Test your code thoroughly before submission to ensure it works as expected.
- Do not use artificial intelligence (AI) tools or external code generation systems. If we catch you, you will be banned from participation.

5 Core Problems (Required)

5.1 Problem 1: Counting Runes (Create Your Own `strlen()`)

Task: Write a function to count the number of characters in a given string. Don't use `strlen()` or other built-in string functions. Create your own version.

Prototype: `int My_strlen(const char *str);`

Explanation: The `My_strlen()` function should count the number of characters in the string `str` until it encounters the null terminator `\0`. The function should return the count of characters excluding the null terminator. Do not use the built-in `strlen()` function.

Main Test Function:

```c
#include <stdio.h>

/**
 * my_strlen - Your Function
 */
int my_strlen(const char *str)
{
    //Your code
}

int main(void)
{
    const char *test1 = "IAE CLUB";
    const char *test2 = "House Of Code";
    const char *test3 = "G";
    const char *test4 = "";

    printf("Test 1: %s\n", test1);
    printf("Length: %d\n", my_strlen(test1));

    printf("Test 2: %s\n", test2);
    printf("Length: %d\n", my_strlen(test2));

    printf("Test 3: %s\n", test3);
    printf("Length: %d\n", my_strlen(test3));

    printf("Test 4: %s\n", test4);
    printf("Length: %d\n", my_strlen(test4));

    return 0;
}
```

Listing 1: Testing My_strlen

Test Cases:

Test 1

Input: s1 = "IAE CLUB" Output: 8

Test 2

Input: s1 = "House Of Code" Output: 15

Test 3

Input: s1 = "G" Output: 1

Test 4

Input: s1 = "" Output: 0

5.2 Problem 2: The Lost Scrolls of Algoria (Reverse a String)

In the ancient kingdom of Algoria, a powerful scroll containing secret knowledge has been written backwards! The scholars need your help to reverse the order of letters so they can read its true meaning.

Task: Write a function that reverses a string without using the built-in string-reversal functions.

Prototype: `void reverse_string(char *str);`

Explanation: You have been given a string, but its characters are in reverse order! Your task is to flip the string so that it reads correctly from left to right. You must modify the original string without using built-in functions like strlen() or strrev()

Tips: Use your my_strlen() from Problem 1 to determine the string's length instead of using strlen().

Main Test Function:

```c
#include <stdio.h>


/**
 * reverse_string - Your Function
 */
void reverse_string(char *str)
{
    //Your code
}

int main(void)
{
    char test1[] = "edoc fo esuoH oT emocleW";
    char test2[] = "uoy pleh lliw ti ;3 melborp ni noitcnuf siht esU";
    char test3[] = "Hello World";
    char test4[] = "G";

    printf("Before: %s\n", test1);
    reverse_string(test1);
    printf("After: %s\n\n", test1);

    printf("Before: %s\n", test2);
    reverse_string(test2);
    printf("After: %s\n\n", test2);

    printf("Before: %s\n", test3);
    reverse_string(test3);
    printf("After: %s\n\n", test3);

    printf("Before: %s\n", test4);
    reverse_string(test4);
    printf("After: %s\n\n", test4);

    return 0;
}
```

Listing 2: reverse_string

Test Cases:

Test 1

Input: s1 = "edoc fo esuoH oT emocleW"

Output: "Welcome To House of code"

Test 2

Input: s1 = "uoy pleh lliw ti ;3 melborp ni noitcnuf siht esU"

Output: "Use this function in problem 3; it will help you"

Test 3

Input: s1 = "Hello World"

Output: "dlroW olleH"

Test 4

Input: s1 = "G"

Output: "G"

5.3 Problem 3: The Whisper of the Ancients (Reverse Words in a String)

In the Kingdom of Valoria, an ancient spell was discovered—one that can reverse the words of a message to reveal hidden truths. However, the spell is broken, and the words appear in the wrong order.
The Grand Maester needs your help to restore the message to its original form.

Task: Write a function that reverses the order of words in a given string.

Prototype: `void reverse_words(char *str);`

tips:

- Use your my_strlen() function from Problem 1 to determine the length of the string.
- Use the string reversal function from Problem 3 to reverse the entire string.
- Be mindful of spaces and ensure that no extra spaces are added in the final result.

Main Test Function:

```c
#include <stdio.h>

/**
 * reverse_string - Your Function
 */
void reverse_words(char str[])
{
    //Your code
}

int main(void)
{
    char test1[] = "The dragons are coming";
    char test2[] = "code love I";
    char test3[] = "G";

    printf("Before: %s\n", test1);
    reverse_words(test1);
    printf("After:  %s\n\n", test1);

    printf("Before: %s\n", test2);
    reverse_words(test2);
    printf("After:  %s\n\n", test2);

    printf("Before: %s\n", test3);
    reverse_words(test3);
    printf("After:  %s\n\n", test3);

    return 0;
}
```

Listing 3: Testing reverse_words

Test Cases:

Test 1

Input: s1 = "The dragons are coming"

Output: "coming are dragons The"

Test 2

Input: s1 = "code love I"

Output: "I love code"

Test 3

Input: s1 = "G"

Output: "G"

5.4 Problem 4: Guardian of the Ancient Scrolls (Valid Parentheses Check)

In the kingdom of **Algoria**, the *Royal Archives* store ancient scrolls written in a special language that uses **( ) { }** [ ] to mark different sections. However, some scrolls have been damaged over time, causing their structure to become unbalanced. The **Royal Archivist** needs your help to determine if a given scroll is properly structured.

Task: Write a function to check whether the brackets `( ) { } [ ]` in a string are correctly written and balanced.

A string is considered **valid** if:

- Every opening bracket (`(`, `{`, `[`) has a matching closing bracket (`)`, `}`, `]`).
- Brackets close in the **correct order** (like in mathematical expressions or programming syntax).

Prototype: `int isValid(const char *str);`

tips:

- You can use a **stack** data structure to solve this problem, but it will increase **space complexity**.
- Alternatively, you can solve it using only the given string (without an extra stack), which will **reduce space complexity**.
- Choose the approach that feels easier for you, but if you want to optimize space, try avoiding an additional stack.

Main Test Function:

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


bool isValid(const char *s)
{
    //Your code here
}

int main(void)
{
    const char *test1 = "()";
    const char *test2 = "[{()}]";
    const char *test3 = "{[(a+b) * x}";
    const char *test4 = "{[a+b]*(x/y)}";

    printf("Test 1: %s\n", test1);
    printf("Is valid: %d\n", isValid(test1));

    printf("Test 2: %s\n", test2);
    printf("Is valid: %d\n", isValid(test2));

    printf("Test 3: %s\n", test3);
    printf("Is valid: %d\n", isValid(test3));

    printf("Test 4: %s\n", test4);
    printf("Is valid: %d\n", isValid(test4));

    return 0;
}
```

Listing 4: Testing isValid

Test Cases:

Test 1

Input: s1 = `"()"`

Output: true

Test 2

Input: s1 = `"[()]"`

Output: true

Test 3

Input: s1 = "{[$a + b)\ \ *\ \ x$}"

Output:$false$

Test 4

Input: s1 = `"{[a+b]*(x/y)}"`

Output: true

6 bonus problem (Advanced - Optional)

6.1 Problem5 (Advanced - Optional): String Splitter (No Built-in Tokenization Functions)

In this problem, you are required to split a string into multiple substrings using a specified delimiter. While you are allowed to use `string.h`, `stdlib.h`, and `stdio.h` libraries, you are **not allowed** to use tokenization functions such as `strtok()`.

Task: Write a function that takes a string and a delimiter character, and splits the string into multiple substrings without using the `strtok()` function. You can use other functions from `string.h` (like `strlen()`, `memcpy()`, etc.) and memory allocation functions from `stdlib.h` (like `malloc()`) to solve the problem.

The function should return an array of dynamically allocated strings. You will need to handle the memory allocation for each substring, and make sure to handle edge cases like:

- Empty strings.
- Delimiters that appear at the beginning or end of the string.
- Consecutive delimiters.

Prototype: `char** split_str(const char* str, char delimiter);`

Tips:

- You can use the `malloc` or `calloc` function to allocate memory for the array of substrings.
- You can use `realloc` to change the size of the array that holds the strings, so it can fit more or fewer strings as needed
- Use a loop to traverse the string and manually split it into substrings when encountering the delimiter.
- Try to use `strlen` and `strncpy` for string length and copying.
- Be sure to handle memory freeing after use to avoid memory leaks.

Main Test Function:

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

char** split_str(const char* str, char delimiter)
{
    // Your code here
}

int main(void)
{
    char * str = "IAE,CLUB,CHALLANGE";
    char **vector = split_str(str, ',');
    int i;

    i = 0;
    while (vector[i] != NULL)
    {
        printf("Substring %d : %s\n",i + 1 ,vector[i]);
        i++;
    }

    while (vector[i] != NULL)
    {
        free(vector[i]);
        i++;
    }

    free(vector);
    return (0);
}
```

Listing 5: Testing splitString

Test Cases:

Test 1

Input: str = "IAE,CLUB,CHALLANGE", delimiter = ','

Output:

- Substring 1: `IAE`
- Substring 2: `CLUB`
- Substring 3: `CHALLANGE`

Test 2

Input: str = `"IAE CLUB"`, delimiter = `' '`

Output:

- Substring 1: `IAE`
- Substring 2: `(empty string)`
- Substring 3: `CLUB`

Test 3

Input: str = `" /IAE/ /CLUB/"`, delimiter = `'/'`

Output:

- Substring 1: `(empty string)`
- Substring 2: `IAE`
- Substring 3: `(empty string)`
- Substring 4: `CLUB`

7 Submission Instructions

Please follow one of the methods below to submit your code:

- **GitHub:** Push your code to a GitHub repository and share the repository link with us.
- **Google Drive:** Upload your code file to Google Drive and share the link with us.
- **Other Methods:** If you are unable to use GitHub or Google Drive, feel free to use another file-sharing service. Ensure that the file is accessible and share the link.

textbfNote: Before submitting, please verify that:

- Your repository or file has the appropriate access permissions.
- You have thoroughly tested your code.

Click here to submit your work.