27.- ft_split.-

Based on BSD Man Pages: Loosely similar to `strsep(3)` and `strtok(3)`, but with different behavior and memory management. **Library Association: “**libft.h”.

Synopsis: `char **ft_split(char const *s, char c);`

Purpose:

- Splits a given string (`s`) into an array of null-terminated substrings, delimited by a specified character (`c`).
- Allocates memory for the resulting array and individual substrings.
- Returns a pointer to the newly created array of strings.

Parameters:

- `s`: The string to be split (const char pointer).
- `c`: The character used as the delimiter (char).

Return Value:

- Returns a pointer to a newly allocated array of null-terminated strings (char **).
- Returns `NULL` if memory allocation fails, invalid input is provided, or no words are found.

Description:

1. **Counts words:** `ft_countwords` iterates through `s`, skipping delimiters and counting non-delimiter characters as individual words.
2. **Allocates memory:** Allocates memory for the array of strings based on the word count plus 1 for the null terminator.
3. **Fills the array:** `ft_fillwords` iterates through `s` again, finding the start and end positions of each word using `ft_countwords`. It then extracts these substrings using `ft_substr` and stores them in the array.
4. **Memory management:** If memory allocation fails at any point, `ft_free` is called to clean up any allocated memory before returning `NULL`. Otherwise, the filled array is returned.

Code:

```
#include "libft.h"

static int	ft_countwords(char const *str, char c)
{
	int	i;
	int	count_words;

	i = 0;
	count_words = 0;
	while (str[i] != '\0')
	{
		while (str[i] == c)
			i++;
		if (str[i] != '\0')
```

```
                count_words++;
            while (str[i] != c && str[i] != '\0')
                i++;
        }
        return (count_words);
}

static void     ft_free(char **str)
{
        int     i;

        i = 0;
        while (str[i])
        {
            free (str[i]);
            i++;
        }
        free (str);
}

static int      ft_fillwords(char const *str, char c, char **matrix)
{
        int     matrix_index;
        int     start;
        int     end;

        matrix_index = 0;
        end = 0;
        while (ft_countwords(str, c) > matrix_index)
        {
            start = end;
            while (str[start] == c)
                start++;
            end = start;
            while (str[end] != c && str[end] != '\0')
                end++;
            matrix[matrix_index] = ft_substr(str, start, (end - start));
            if (!matrix[matrix_index])
            {
                ft_free(matrix);
                return (0);
            }
            matrix_index++;
        }
        return (1);
}

char    **ft_split(char const *s, char c)
{
        char    **matrix;

        if (!s)
```

```
		return (NULL);
	matrix = malloc(sizeof(char *) * (ft_countwords(s, c) + 1));
	if (!matrix)
		return (NULL);
	matrix[ft_countwords(s, c)] = NULL;
	if (ft_fillwords(s, c, matrix) == 0)
		return (NULL);
	return (matrix);
}
/*
int	main(void)
{
	char	*s = "Hola, mundo, hola";
	char	**arr;
	int		i;

	arr = ft_split(s, ',');
	if (arr == NULL)
	{
		printf("Error allocating memory for array\n");
		return (1);
	}
	i = 0;
	while (arr[i])
	{
		printf("Subcadena %d: %s\n", i, arr[i]);
		i++;
	}
	i = 0;
	while (arr[i])
	{
		free(arr[i]);
		i++;
	}
	free(arr);
	return (0);
}
*/
```

Code Explanation:

1. `ft_countwords`**:**

- This function counts the number of words in a string separated by a delimiter (`c`).
- It loops through the string character by character:
  - Skips over any occurrences of the delimiter `c`.
  - If it encounters a non-delimiter character after a delimiter, it increments the `count_words` counter.
- Finally, it returns the total number of words found.

2. `ft_free`**:**

- This function frees the memory allocated to an array of strings.

- It loops through the array and calls `free` on each individual string pointer to release the memory.
- Finally, it frees the memory allocated to the array itself.

3. `ft_fillwords`**:**

- This function fills an array of strings with substrings separated by a delimiter (`c`).
- It uses `ft_countwords` to determine the number of words needed in the array.
- It then loops through the string, finding the start and end of each word:
    - Skips over leading delimiters.
    - Finds the end of the word by looking for the next delimiter or the end of the string.
    - Uses `ft_substr` to extract the substring and allocate memory for it.
    - Stores the extracted substring in the next available slot in the array.

Code Snippet:

```
start = end;
while (str[start] == c)
   start++;
end = start;
```

Explanation:

This code block is part of the `ft_fillwords` function, which is responsible for filling an array of strings with substrings separated by a delimiter (`c`). Specifically, it's within a loop that iterates over each word to be extracted.

Breakdown:

- `start = end;`:
    - At the beginning of each iteration, `start` is set equal to `end`. This is because `end` holds the index of the character after the previous delimiter or the end of the string. Setting `start` to the same value ensures we start searching for the current word from the correct position.
- `while (str[start] == c)`:
    - This `while` loop iterates as long as the character at the current position (`str[start]`) is equal to the delimiter `c`.
    - Inside the loop, `start` is incremented (`start++`). This effectively skips over any leading delimiters before the actual word starts.
- `end = start;`:
    - Once the `while` loop finishes (meaning we encountered a non-delimiter character), `end` is set equal to `start`. This marks the starting position of the actual word.

In essence, this code block skips over any leading delimiters before a word and identifies the starting position of the word within the string.

Additional Points:

- Remember that `start` and `end` are indices, not the actual characters. Their values correspond to the positions within the string array.
- This code assumes that there won't be consecutive delimiters within a word.

- If any memory allocation fails during the process, it frees any already allocated memory using `ft_free` and returns 0.
- Otherwise, it returns 1 to indicate success.

4. `ft_split`**:**

- This is the main function that splits a string into an array of words separated by a delimiter (`c`).
- It first checks if the input string is NULL and returns NULL if it is.
- It allocates memory for an array of string pointers, one more than the number of words (to store a NULL terminator).
- It sets the last element of the array to NULL.
- It calls `ft_fillwords` to populate the array with substrings.
- If `ft_fillwords` fails, it frees the allocated memory and returns NULL.
- Otherwise, it returns the filled array of strings.

  In the line `matrix[ft_countwords(s, c)] = NULL;` within the `ft_split` function, there are two key aspects to understand:

1. Adding a NULL terminator:

- `matrix` is an array of strings that will store the split words.
- `ft_countwords(s, c)` calculates the number of words in the string `s` based on the delimiter `c`.
- By setting `matrix[ft_countwords(s, c)] = NULL;`, the code is adding a NULL pointer (`NULL`) as the last element of the array.
- This NULL pointer acts as a terminator, signifying the end of the valid elements in the array.

2. Importance of the NULL terminator:

- C strings are character arrays terminated by a NULL character (`\0`).
- When iterating through or using string functions on an array of strings, relying on the NULL terminator is crucial to determine the end of each string and avoid accessing invalid memory.
- In this case, subsequent functions (like printing or freeing memory) that process the `matrix` can rely on the NULL terminator to know where the strings end and stop iterating correctly.

In summary, this line ensures that the `matrix` **array has a proper NULL terminator, which is essential for working with C strings safely and efficiently.**

Comments for the `main` **Function:**

The `main` function demonstrates how to use `ft_split` to split a string and iterate through the resulting array. Remember to free the allocated memory using `free` once you're done with the array.

Key Points:

- `ft_split` allocates memory for the array and each substring within it. Remember to `free` the entire array using `free(arr)` when you're finished.
- It handles `NULL` inputs gracefully, returning `NULL` if either `s` or `c` is `NULL`.
- It considers strings with consecutive delimiters at the beginning or end as empty words.

Static in `ft_countwords`, `ft_free`, and `ft_fillwords`:

- In these helper functions, `static` restricts their scope to the current file (where they are defined). This means they are not accessible from other files in the project, even if they are declared in the same header file.
- **Benefits:**
    - Promotes encapsulation and modularity by hiding internal implementation details from other parts of the code.
    - Prevents accidental usage from other files, reducing potential naming conflicts and unintended side effects.
- **Drawbacks:**
    - Limits reusability across different files in the project.

Considerations:

- The choice of using `static` for these helper functions depends on your specific project needs and design approach.
- If these functions are only used internally by `ft_split` and have no need to be visible outside the file, `static` can be beneficial.
- However, if you plan to reuse these functions in other parts of the project, consider using the regular (non-static) approach.

Note: `ft_split` itself is not declared as `static`, meaning it can be accessed and used from other files as long as the header file is properly included.

I hope this clarifies the usage of `static` in the `ft_split` code. If you have further questions or need more specific examples, feel free to ask!

Scope and Lifetime:

- When you declare a variable with `static` inside a function, it stays in memory throughout the entire program's execution, but its scope remains limited to that function. It's essentially hidden from other functions.
- Unlike automatic variables (declared without `static` or `register`), which are created and destroyed each time the function is called, static variables persist even after the function returns.

Memory Management:

- Static variables are allocated on the program's data segment, not the stack. This means they don't get automatically deallocated when the function returns, like automatic variables on the stack.
- However, this doesn't imply they never get freed. The memory allocated for static variables is released when the entire program terminates.

Important Points:

- Using `static` doesn't prevent you from explicitly freeing the memory of static variables if needed within the function using `free`. This is especially relevant if the static variable stores dynamically allocated data like pointers.
- While static variables persist throughout the program, they are still subject to the program's overall memory management. If your program uses excessive static variables or doesn't manage memory properly, it can still lead to memory issues.

In summary:

- `static` mainly controls the variable's scope and visibility, not its lifetime or memory management.
- Static variables persist, but they are released when the program terminates.
- You can free them within the function if needed, but be careful not to introduce memory leaks.