7. They say we only use a fraction of our brain's true potential. Now that's when we're awake. When we're asleep, we can do almost anything
#advanced
Write a function that returns 1 if a string is a palindrome and 0 if not.

Prototype: int is_palindrome(char *s);
An empty string is a palindrome

```
julien@ubuntu:~/0x08. Recursion$ cat 100-main.c
#include "main.h"
#include <stdio.h>

/**
* main - check the code
*
* Return: Always 0.
*/
int main(void)
{
int r;

r = is_palindrome("level");
printf("%d\n", r);
r = is_palindrome("redder");
printf("%d\n", r);
r = is_palindrome("test");
printf("%d\n", r);
r = is_palindrome("step on no pets");
printf("%d\n", r);
return (0);
}
julien@ubuntu:~/0x08. Recursion$ gcc -Wall -pedantic -Werror -Wextra -std=gnu89 100-main.c 100-is_palindrome.c -o 100-palindrome
julien@ubuntu:~/0x08. Recursion$ ./100-palindrome
1
1
0
1
julien@ubuntu:~/0x08. Recursion$
```

my solution

```
#include "main.h"
/**
* s_length - checks the length of a string
* @c: is the string
* Return: return the leng of the string
```

```
**/
int s_length(char *c)
{
if (*c == '\0')
return (0);
return (1 + s_length(c + 1));
}

/**
* check - checks if the string is palindrome
* @x: the index
* @l: the length of the string
* @s: the string
* Return: 1 if is polindrome or 0 if not
**/
int check(int x, int l, char *s)
{
if (l > 0)
{
if (s[x] == s[l])
{
return (check(x + 1, l - 1, s));
}
else if (s[x] != s[l])
{
return (0);
}
else
{
return (1);
}
}
return (1);
}

/**
* is_palindrome - a function that returns 1 if a string is
* a palindrome and 0 if not.
* @s: Array of char
* Return: return 1 if the string is a palindrome or 0 otherwise
**/
int is_palindrome(char *s)
{
return (check(0, s_length(s) - 1, s));
}
```

breack down

It seems you've provided a solution that checks if a given string is a palindrome using recursion. Your solution breaks down the problem into three functions:

1. **s_length:** Finds the length of the given string recursively.
2. **check:** Checks if the string is a palindrome by comparing characters from the start and the end of the string, moving inwards until all characters are checked.
3. **is_palindrome:** The main function that integrates the above two.

Here's a breakdown of each function:

1. s_length:

The function `s_length` recursively determines the length of a string.

- **Base Case**: If the current character is `\0` (null terminator), it returns 0.
- **Recursive Case**: For each character, it adds 1 to the count and moves to the next character by recursively calling itself.

2. check:

The function `check` uses a recursive approach to determine if the string is a palindrome.

- **Parameters**:
  - **x**: Current starting index.
  - **l**: Current ending index.
  - **s**: The string.
- **Recursive Logic**:
  - It compares characters from the start and the end of the string.
  - If they are the same, it continues the recursion by comparing the next set of characters (i.e., `x + 1` and `l - 1`).
  - If they are different, it immediately returns 0 (indicating it's not a palindrome).
  - If `l > 0` is not satisfied, it means we've checked all characters and can conclude that it's a palindrome.

3. is_palindrome:

This is the primary function called in the main program. It sets up the initial indices for the `check` function and calls it. It also determines the length of the string using `s_length`.

Points to Note:

1. The solution does not consider cases where there might be spaces or punctuation in the string (e.g., "step on no pets" would not work even though the output you showed is 1). It compares the strings directly.
2. In a more comprehensive solution, you'd likely want to strip spaces and punctuation and perhaps even make the comparison case-insensitive.

Your solution seems valid for simple strings without spaces or punctuation, and it uses a recursive approach effectively. If you need it to handle cases like "step on no pets", then additional preprocessing steps will be needed before checking for palindromes.

breack it down with ex

Certainly, let's go through your solution with detailed explanations and examples:

1. s_length:

This function determines the length of the string.

```
int s_length(char *c)
{
	if (*c == '\0')
		return (0);
	return (1 + s_length(c + 1));
}
```

Example: Input: `c = "level"` Output: `5`

How it works:

- It starts with the first character.
- If it's not the end of the string (`\0`), it adds 1 and moves to the next character.
- The function continues to count characters until it hits the end of the string.

2. check:

This function checks if the string is a palindrome.

```
int check(int x, int l, char *s)
{
	if (l > 0)
	{
		if (s[x] == s[l])
		{
			return (check(x + 1, l - 1, s));
		}
		else if (s[x] != s[l])
		{
			return (0);
		}
		else
		{
			return (1);
		}
	}
	return (1);
}
```

Example: Input: `x = 0`, `l = 4`, `s = "level"` How it works:

- Compares `s[0]` (which is `l`) with `s[4]` (which is also `l`). They're the same.
- Continues by checking `s[1]` and `s[3]`. They're also the same.
- Doesn't need to check `s[2]` since we're in the middle.
- Returns `1`, indicating it's a palindrome.

3. is_palindrome:

This is the main function that sets up and checks if the string is a palindrome.

```
int is_palindrome(char *s)
{
        return (check(0, s_length(s) - 1, s));
}
```

Example: Input: `s = "level"` How it works:

- Calls `s_length` to get the length, which returns `5`.
- Calls `check` with `x = 0` and `l = 4`.
- `check` confirms it's a palindrome and returns `1`.

Step by Step:

Let's walk through the process using the string "level".

1. `is_palindrome` is called with "level".
2. Inside `is_palindrome`, `s_length` is called to find the length of "level" which is 5.
3. `check` is then called with parameters `x = 0`, `l = 4`, and `s = "level"`.
4. Inside `check`, `s[0]` (which is `l`) and `s[4]` (which is also `l`) are compared. Since they're the same, the function recurses.
5. Now, `check` is called with `x = 1` and `l = 3`. This time `s[1]` (which is `e`) and `s[3]` (which is also `e`) are compared. They're the same again, so the function recurses.
6. On the next call, `x` would be `2` and `l` would be `2`. At this point, we've reached the center of the string, so without further checking, it returns `1` indicating "level" is a palindrome.

The solution uses recursion to methodically compare characters from the outermost part of the string inwards, verifying if the string is a palindrome.

From ChatGPT, designed with FancyGPT