Write a function that returns the length of a string.

Prototype: int _strlen_recursion(char *s);
FYI: The standard library provides a similar function: strlen. Run man strlen to learn more.

julien@ubuntu:~/0x08. Recursion$ cat 2-main.c
#include "main.h"
#include <stdio.h>

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

n = _strlen_recursion("Corbin Coleman");
printf("%d\n", n);
return (0);
}
julien@ubuntu:~/0x08. Recursion$ gcc -Wall -pedantic -Werror -Wextra -std=gnu89 2-main.c 2-strlen_recursion.c -o 2-strlen_recursion
julien@ubuntu:~/0x08. Recursion$ ./2-strlen_recursion
14

my solution

```
int _strlen_recursion(char *s)
{
int count = 0;

if (*s != '\0')
{
_strlen_recursion(s + 1);
count++;
}
return (count);
}
```

The function you provided seems to be on the right track, using recursion to traverse through the string. However, there's a problem with the function: you are not accumulating the counts during the recursive calls.

Here's the corrected version of your function:

```
int _strlen_recursion(char *s)
{
    if (*s == '\0')
        return 0;
    else
        return 1 + _strlen_recursion(s + 1);
}
```

In this corrected function:

- If the current character is the null terminator (`\0`), the function returns `0` since there's nothing more to count.
- Otherwise, it counts `1` for the current character and adds the result of the recursive call to the next character (using `s + 1`).

With this function, the recursion correctly accumulates the lengths of the string until the base case is met (when `*s` is `\0`).

without the solution

Certainly! Let's diagnose the problem step-by-step.

First, let's break down the logic you've written:

```
int _strlen_recursion(char *s)
{
    int count = 0;

    if (*s != '\0')
    {
        _strlen_recursion(s + 1);
        count++;
    }
    return (count);
}
```

1. You're initializing `count` to `0`.
2. If the current character is not the null terminator, you call the function again with the next character in the string.
3. After the recursive call, you increment `count` by `1`.
4. Finally, you return `count`.

There's an issue here. In the recursive process, you are correctly moving to the next character and incrementing the count. But the problem is the `count` variable is local to each function call. So, each time you recursively call `_strlen_recursion`, a new `count` variable is created and set to `0`. When the recursion unwinds and returns, each function call only returns `1` or `0`, but you never accumulate the total count.

**Hint**: You need to combine the `count` of the current function call with the result of the recursive call.

Can you spot the change needed based on the hint?

return 1 + _strlen_recursion(s + 1);
don't under

Of course! Let's understand this step by step.

The core idea behind the expression `return 1 + _strlen_recursion(s + 1);` is to account for the length of the current character while also counting the remaining characters in the string.

Here's a breakdown:

1. `1`: This represents the current character in the string. Each character has a length of 1. So for every character that is not the null terminator, we're adding `1` to the total length.
2. `_strlen_recursion(s + 1)`: This is a recursive call to the function for the rest of the string, i.e., starting from the next character. The expression `s + 1` advances the pointer to the next character in the string.

Let's take a simple example: If our string is `"hi"`, the process will be:

1. First call: `s` points to `"hi"`
    2. `*s` is `'h'`, so we count `1` for this character.
    3. We then make a recursive call with `s + 1` which points to the next character, `"i"`.
4. Second call: `s` points to `"i"`
    5. `*s` is `'i'`, so again we count `1` for this character.
    6. We then make a recursive call with `s + 1` which now points to the null terminator.
7. Third call: `s` points to the null terminator (`"\0"`)
    8. Here, the base case applies, so we return `0` because we don't count the null terminator.

Now, as we return from the recursive calls:

- The third call returns `0`.
- The second call returns `1 (for 'i') + 0 (from the third call) = 1`.
- The first call returns `1 (for 'h') + 1 (from the second call) = 2`.

So, the total length returned for the string `"hi"` is `2`.

This "accumulate as we return" approach ensures that for every character in the string, we're adding `1` to our total length.