

Write a function that adds a new node at the end of a list_t list.

Prototype: list_t *add_node_end(list_t **head, const char *str);

Return: the address of the new element, or NULL if it failed

str needs to be duplicated

You are allowed to use strdup

julien@ubuntu:~/0x12. Singly linked lists\$ cat 3-main.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "lists.h"

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

    head = NULL;
    add_node_end(&head, "Anne");
    add_node_end(&head, "Colton");
    add_node_end(&head, "Corbin");
    add_node_end(&head, "Daniel");
    add_node_end(&head, "Danton");
    add_node_end(&head, "David");
    add_node_end(&head, "Gary");
    add_node_end(&head, "Holden");
    add_node_end(&head, "Ian");
    add_node_end(&head, "Ian");
    add_node_end(&head, "Jay");
    add_node_end(&head, "Jennie");
    add_node_end(&head, "Jimmy");
    add_node_end(&head, "Justin");
    add_node_end(&head, "Kalson");
    add_node_end(&head, "Kina");
    add_node_end(&head, "Matthew");
    add_node_end(&head, "Max");
    add_node_end(&head, "Michael");
    add_node_end(&head, "Ntuj");
```

```
        add_node_end(&head, "Philip");
        add_node_end(&head, "Richard");
        add_node_end(&head, "Samantha");
        add_node_end(&head, "Stuart");
        add_node_end(&head, "Swati");
        add_node_end(&head, "Timothy");
        add_node_end(&head, "Victor");
        add_node_end(&head, "Walton");
        print_list(head);
        return (0);
    }
}
```

```
julien@ubuntu:~/0x12. Singly linked lists$ gcc -Wall -pedantic -Werror -
Wextra -std=gnu89 3-main.c 3-add_node_end.c 0-print_list.c -o d
```

```
julien@ubuntu:~/0x12. Singly linked lists$ ./d
```

```
[4] Anne
[6] Colton
[6] Corbin
[6] Daniel
[6] Danton
[5] David
[4] Gary
[6] Holden
[3] Ian
[3] Ian
[3] Jay
[6] Jennie
[5] Jimmy
[6] Justin
[6] Kalson
[4] Kina
[7] Matthew
[3] Max
[7] Michael
[4] Ntuj
[6] Philip
[7] Richard
[8] Samantha
[6] Stuart
[5] Swati
[7] Timothy
[6] Victor
[6] Walton
```

```
julien@ubuntu:~/0x12. Singly linked lists$
```

```
Repo:
```

GitHub repository: [alx-low_level_programming](#)

Directory: 0x12-singly_linked_lists

File: 3-add_node_end.c

my solution

```
list_t *add_node_end(list_t **head, const char *str)
{
    list_t *end = malloc(sizeof(list_t));
    if (!end)
        return (NULL);

    if (!str)
    {
        end->str = NULL;
        end->len = 0;
    }
    else
    {
        end->str = strdup(str);
        if (!end->str)
        {
            free(end);
            return (NULL);
        }
        end->len = strlen(str);
    }
    end->next = NULL;

    if (!*head)
    {
        *head = end;
    }
    else
    {
        list_t *strNode = *head;
        while (strNode->next)
            strNode = strNode->next;
        strNode->next = end;
    }

    return (end);
}
```

explain this task line by line full details no summary

Sure, let's go through your `add_node_end` function line by line to understand its functionality in detail. This function is designed to add a new node at the end of a singly linked list.

1 Function Definition:

```
list_t *add_node_end(list_t **head, const char *str)
```

This line defines the function `add_node_end`. It returns a pointer to `list_t`, which is the type of the nodes in your linked list. The function takes two parameters:

- 2 `head`: a double pointer to `list_t`, which points to the head of the list.
- 3 `str`: a constant string pointer representing the data that needs to be stored in the new node.

4 Creating a New Node:

```
list_t *end = malloc(sizeof(list_t));  
if (!end)  
    return (NULL);
```

Here, you allocate memory for a new list node (`end`) using `malloc`. The size of memory allocated is equal to the size of `list_t`. If `malloc` fails to allocate memory (i.e., returns `NULL`), the function immediately returns `NULL`, indicating failure.

5 Handling Null String Input:

```
if (!str)  
{  
    end->str = NULL;  
    end->len = 0;  
}
```

These lines handle the case where the input string `str` is `NULL`. In this case, you set the `str` field of the new node to `NULL` and its length (`len`) to 0.

6 Copying the String:

```
else  
{  
    end->str = strdup(str);  
    if (!end->str)  
    {  
        free(end);  
        return (NULL);  
    }  
    end->len = strlen(str);  
}
```

If `str` is not `NULL`, you use `strdup` to duplicate the string and store the duplicate in the new node's `str` field. If `strdup` fails (i.e., returns `NULL`), you free the previously allocated memory for the node and return `NULL`. If successful, you store the length of the string in the `len` field.

7 Initializing the Next Pointer:

```
end->next = NULL;
```

This sets the `next` pointer of the new node to `NULL`, as it will be the last node in the list.

8 Adding the Node to the List:

```

if (!*head)
{
    *head = end;
}
else
{
    list_t *strNode = *head;
    while (strNode->next)
        strNode = strNode->next;
    strNode->next = end;
}

```

These lines add the new node to the end of the list. If the list is currently empty (**head* is *NULL*), you set **head* to point to the new node. Otherwise, you traverse the list starting from *head* until you find the last node (whose *next* pointer is *NULL*) and set its *next* pointer to the new node (*end*).

9 Returning the New Node:

```

return (end);

```

Finally, you return a pointer to the new node.

This function effectively adds a new node to the end of a singly linked list, duplicating the given string in the process. The use of *strdup* simplifies string handling but requires care to ensure that memory is properly managed, especially in error cases.

```

if (!*head)
{
    *head = end;
}

```

why hear we assin end to *header

In the line:

```

if (!*head)
{
    *head = end;
}

```

This part of the code is handling the situation where your linked list is initially empty. Let's break it down:

- 1 if (!*head):** This is checking if the pointer to the first node of the list (which is pointed to by **head*) is *NULL*. In other words, it's checking if the list is empty. The **head* dereferences the

double pointer to get the actual head pointer of the list. The `!` operator negates the result, so `!*head` is `true` if `*head` is `NULL`.

- 2 `*head = end;` If the list is indeed empty (`*head` is `NULL`), this line sets the head of the list to the newly created node. Here, `end` is a pointer to the new node that you have just created and initialized. Since `head` is a pointer to a pointer (double pointer), you need to dereference it once (`*head`) to modify the actual head pointer of the list. By assigning `*head = end;`, you're saying that the head of the list now points to the new node `end`.

Essentially, when the list is empty, the new node you create becomes the first (and only) node in the list. That's why you directly assign `end` to `*head` in this case. If you didn't do this, there would be no way to access the new node using the head of the list, as the list's head would still be `NULL`, effectively making the new node disconnected from the list.

`return (end);`

`end` is the final node how we return it instead of `strNode`

Returning `end` in the `add_node_end` function, instead of returning the previous final node or the head node (like `strNode`), is a design choice that depends on the requirements of your function and how it will be used. Let's examine why returning `end` can be a useful and appropriate choice:

- 1 **Function Purpose:** The primary purpose of `add_node_end` is to add a new node at the end of the list. By returning the newly added node (i.e., `end`), the function provides immediate access to this node. This can be helpful if the caller needs to perform further operations specifically on this new node, such as additional setup or linking it to other structures.
- 2 **Confirmation of Success:** Returning the new node (`end`) also serves as a confirmation that the node was successfully added to the list. If the function returns `NULL`, it indicates an error occurred (such as memory allocation failure). This makes error handling easier for the caller.
- 3 **Consistency with Other Functions:** If you have other functions in your linked list implementation that also return the node they work with (like an `add_node_beginning` function), returning the `end` node here maintains consistency across your API. This consistency can make the library easier to understand and use.
- 4 **Avoiding Redundancy:** Returning the head of the list (`*head`) or the previous last node (`strNode`) isn't usually necessary because the caller already has access to the head of the list (since it's passed as an argument to the function), and the previous last node is not as commonly needed immediately after adding a new node to the end of the list.
- 5 **Flexibility:** If the caller of the function does need access to the head of the list or any other node, they already have it or can easily traverse the list to find it. Returning the new end node does not restrict them from accessing other parts of the list.

In summary, returning the newly added node (`end`) is a practical choice that offers immediate access to this node for further operations, confirms successful addition, and aligns with a consistent and flexible API design.

what is the diff between `strNode` and `end`

In your `add_node_end` function, `strNode` and `end` serve different purposes within the context of a singly linked list. Here's a breakdown of their differences:

- 1 `strNode`:
 - 2 **Purpose:** `strNode` is used as a temporary pointer for traversing the list. Its primary role is to find the current end of the list so that the new node can be appended to it.
 - 3 **Usage in the Code:** When adding a new node to a non-empty list, `strNode` starts at the head of the list (`*head`) and moves through each node by following the `next` pointers. It stops when it reaches a node whose `next` pointer is `NULL`, indicating the end of the list.
 - 4 **Lifetime:** The scope of `strNode` is limited to the `else` block where it is used for traversal. Once the new node is linked to the list, `strNode` is no longer needed.
- 5 `end`:
 - 6 **Purpose:** `end` represents the new node that is being added to the list. It is the main focus of the function.
 - 7 **Usage in the Code:** `end` is created using `malloc` to allocate memory for the new node. It is initialized with the provided string (`str`) and its `next` pointer is set to `NULL`. This node is then linked to the list, either as the first node (if the list was empty) or appended to the current last node (found by `strNode`).
 - 8 **Lifetime:** The `end` node becomes a permanent part of the list once it is linked. Its memory remains allocated as part of the list structure, and it is the node returned by the function.

In summary, `strNode` is a temporary pointer used to find the last node of the list, while `end` is the new node that is being added to the list. The function traverses the list using `strNode` and appends `end` at the list's end, returning `end` as the result of this operation.

example

Certainly! Let's go through an example to illustrate how `strNode` and `end` function within the context of adding a node to the end of a singly linked list. Suppose you have a simple singly linked list and you want to add a new node to its end.

Initial List

Imagine your list currently has three nodes with the values "A", "B", and "C". It looks like this:

```
head -> [A] -> [B] -> [C] -> NULL
```

Here, each box `[]` represents a node, and the arrows represent the `next` pointers that link the nodes together. The `head` pointer points to the first node ("A"), and the last node ("C") points to `NULL`, indicating the end of the list.

Adding a New Node

Now, let's say you want to add a new node with the value "D" to the end of the list using your `add_node_end` function.

- 1 **Creating the New Node (`end`):**
 - 2 You allocate memory for a new node and store the value "D" in it. This new node is `end`.
 - 3 Initially, it looks like this: `[D] -> NULL`.
- 4 **Finding the End of the List (`strNode`):**
 - 5 You need to find where to insert this new node. That's where `strNode` comes in.
 - 6 `strNode` starts at the head of the list (`head`) and traverses the list to find the end. It steps through the nodes one by one.
- 7 **Traversal:**
 - 8 First, `strNode` points to "A", then moves to "B", and then to "C".
 - 9 When `strNode` is at "C", it finds that `strNode->next` is `NULL`, indicating the end of the list.
- 10 **Appending the New Node:**
 - 11 Now, you append `end` to the list by setting the `next` pointer of "C" (where `strNode` currently points) to `end`.
 - 12 The list now looks like this: `head -> [A] -> [B] -> [C] -> [D] -> NULL`.
- 13 **Returning the New Node:**
 - 14 The function returns `end`, which is the node containing "D".

Final List

After the function call, the list is:

```
head -> [A] -> [B] -> [C] -> [D] -> NULL
```

And the new node (`end`) containing "D" is successfully added to the end of the list. `strNode` was used just to find the right place to insert `end` and does not have any role after the insertion is complete.

From ChatGPT, designed with  FancyGPT