# Module [202] – Practice Assignment

Wouldn't it be great to have a function which can take a string, identify the words in it, then apply a given function to each word and provide us with the modified original string? Yes, it would. As an extraordinary coincidence, this is exactly what we are going to do in this programming assignment.

## The files you will have to work with

You will be provided with several files to get you started working on this assignment. Make sure you edit them but do not alter their name or add new files. Here are the files;

| | |
|---|---|
| *main.c* | do not modify this file, it will by default call the two functions you have to implement and apply them to a list of tests which you will also define. |
| *tools.h* | do not modify this file; it provides the headers needed to use our functions in other files. |
| *tools.c* | Part of your work will consist in implementing two functions in this file. The prototypes are already provided along with comments describing what they should be doing and a "mock implementation" which makes them compile and do something not so useful for now. |
| *tests.c* | The other part of your work will consist in devising tests which will guarantee that your implementations are valid. Follow the guidelines on how to design good tests harnesses and submit them for credit. |

## Business as usual

By now, you should be used to the fact that you need to implement a solution to the requirements expressed in these instructions, develop tests, posts and discuss your tests on the module's forum and refactor here and then to improve the quality of your solution.

# Task #1 – Implementing and testing word_reverse

The first of the two functions you have to implement is meant to reverse a single word inside a string.

## Some reading before you start

Before to start working on implementing anything, you'll have to read the entire source and make sure you understand what it is doing so that you are able to add to it without breaking everything.

Pay specific attention to the way the program parts already written for you will use the data in **tests_inputs** and **tests_expected**, defined in *tests.c*, in order to invoke your function with some pre-determined parameters and verify it returns the expected result.

Make sure you also understand that the word_reverse function is invoked from **words_modify** which, in turn, pre-processes the data string to work on with **words_initialize**. This is why you are provided with a temporary implementation for **words_initialize**. It is meant to do just enough to enable **words_modify** to invoke your **word_reverse** function so that you may test it.

More specifically, **words_initialize** only assumes that the string we will be working on hold only a single word without spaces before or after it. This is enough in so far that we want to test our **word_reverse** function on single words first before to move to task #2.

In addition, spend some time reading the requirements in this document and modify the arrays in *tests.c* to reflect the tests you think your program ought to pass to meet these requirements. When you do so, make sure you update accordingly the value of the **NB_TESTS** integer constant defined in the same file.

The most observant among you will have noticed that the **main** function uses **strdup**. This is a tool we are going to use more and more in our assignment. This function takes a string and returns a copy of it. To do so it uses dynamical memory allocation which we will study in module [203]. For now, it is enough for you to understand that anything allocated by your own program or a function you are using will need to be de-allocated. This is why we use the function named **free**. More details on this in next module.

When you think the tests you wrote based only on the requirements are sufficient, post this preliminary version in a new thread on the module's forum.

## Implementing your solution

The **word_reverse** function operates on a string which is assumed to hold a single word. It will take all its characters, except the end-of-string marker '\0', and reverse their order. You will reverse the order in the string itself, by swapping characters two by two.

## Testing your solution

You will then validate your solution by providing some tests. These tests, since they will be ran for the entire program, will have to be designed to test only reversing strings with one word in them. The "mock implementation" that is provided to you for the other function you have to write, **words_initialize**, will make it so that the program assumes

your string has a single word anyway. As you add or remove tests in *tests.c*, make sure that you update the **NB_TESTS** integer constant.

### Having others do the testing for you

As for the previous PA, time to go on the module's forum and download other students' testing files to help you find bugs.

### Bit of refactoring

Same as the previous PA, time to improve a bit your program now that it works

## Task #2 – Implementing and Testing words_initialize

The second function you will have to implement will be the one actually locating the beginning of each word in the string provided as parameter **str** to it. In order to help you determine which characters are "separators" between words, a function **is_separator** is provided to you. Do not worry about its implementation; we'll detail it in module [203].

### Implementing your solution

The logic to identify words is to

- Start at the beginning of your string and skip any character which his recognized by **is_separator** as a separator.
- The first non-separator character you meet is the beginning of your first word.
- You now keep skipping all non-separator characters which follow until we meet either the end of string marker '\0' or a separator.
- When we do, we now know where the first word ends and we replace this character by '\0' to end the substring for this word.
- If we didn't meet the end of string marker yet, we now skip all following separators until… you guessed it; the beginning of the next word.

The address of the first character of each word will be stored in an array of pointers which is also passed as parameter to your function. This array has a maximal size, also a parameter, which means that your function should not drop identifying any further words in the string if it already reached the maximal number of words it is able to store in the pointer array.

So, in summary, you iterate over the string, find the first character of each word and store its address at the right index in your array parameter named **words**. Then you change the character immediately following the last character of each word so that it becomes a '\0'.

Together, these two actions will result in you having the address of the first character of every word in your pointers array. When we will use this address, we will be able to display or process each word as if it was its own string. This is due to the fact that we terminated each segment of the original string containing a word with a '\0'.

Later, after we are doing performing some processing on each of these substrings, we will be removing these '\0' and replacing them by spaces. This is not for you to do; the function **words_modify** will do that for you by invoking **word_handle_marker** after you are done with your parts.

## Testing your solution

Add new tests for this implementation to the previous series of tests; now you're able to evaluate how your functions handle sentences.

Whereas your initial tests used only a single word, you may now use entire sentences with spaces between words, maybe multiple ones, maybe punctuation marks to see how your program behaves.

## Having others do the testing for you

Post your tests and discuss with other students if your observations are right or if some of your tests revealed bugs.

## Bit of refactoring

You know the drill by now…