M5PA
Word by Word

Synopsis

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

Files you will be working with

You will be provided with several files to get you started working on this assignment.

- You must not alter the file names, remove or add files to the project
- You must only modify the ones marked below with a yes in "Modify it?"

Here are the files;

File name	Modify it?	Role
tools.c	**Yes**	Implementation of your solution to the assignment
tests.c	**Yes**	Implementation of your test functions
tools.h	**No**	Header file for *tools.c*
main.c	**No**	Implementation of the main function starting your tests
testlib.h	**No**	Definition of the *TEST* function you must use in your tests
testlib.c	**No**	Implementation of the above

Task #1 – Implementing and testing word_reverse

The first function to implement is meant to reverse a single word inside a string.

Some reading before you start

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

In addition, spend some time reading the requirements in this document and add some test functions in *tests.c* to reflect the tests you think your program ought to pass to meet these requirements.

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 write tests to validate your solution. For task #1, we are ready to test only the reversal of single words strings. The "mock implementation" provided to you for **words_initialize** will make it so that the program assumes your string has a single word.

Please note that the functions *words_modify* and *words_reverse* both modify the strings that are passed to them. If you use them and specify as parameter a literal string, things will not work. Instead, you should always call them with a dynamically allocated copy of a string as in the example below;

```
char* stringParameter = strdup("hello");
words_modify(stringParameter );
```

strdup is a tool we are going to use more and more in our assignments. This function takes a string and returns a copy of it. To do so it uses dynamical memory allocation which we introduced in this module.

Similarly, you will have to use *strcmp* to test if two strings are the same. There is no way to do so using the == operator as you would to compare two integers.

```
char* stringParameter = strdup("hello");
words_modify(stringParameter );
int result = strcmp(stringParameter, "olleh");
TEST("reversing [hello]", result == 0);
```

If you have questions about these two string manipulation functions, take a look at your textbook's index and ask your instructor for help.

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 that 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.