Technische Universität München

Informatics I for MSE

MSE IN8011
Prof. Bjoern Menze, Giles Tetteh
Philipp Kaeß

WS 19/20
Homework 3

Due Date: 24.11.19, 23:59

Release Date: 15.11.19

Homework 3: String Handling

Topics: Strings, `Char`, Arrays, Functions, Arrays as Function Parameters

Introduction

What is a `char`? This may seem like a basic question, but is in fact a little bit more complex. First of all, `char` is just like `int` a datatype where you can store whole numbers. There are only two differences between an `int` and a `char`: A `char` needs a smaller amount of memory, and therefore has of course also a smaller range of values than an `int`. And secondly, the numerical value of a `char` **can** represent a letter.

In order to be able to represent and store letters, a system called ASCII has been introduced. This is basically just a table, where each letter is signed to a value of the range of a `char`. E.g. the value `97` is signed to the letter '`a`'. Imagine a new `char` with the value `97` being created. There are now two possible ways to interpret this `char`: One could simply interpret it as a smaller version of `int`, and do calculations with it, e.g. add it to another `char` and do various `int`-calculations. Or one could interpret it as a representation of the letter '`a`'. When printing the `char`-variable, we can decide which interpretation we want to use: If we print the `char` with `%d` as format specifier, '`97`' will be printed. But if we use `%c` instead, we will print '`a`'. You can also directly assign a letter to a `char` variable, but it is important for you to know that in the background the letter is converted to the respective ASCII-code, and then this number is stored. This is the reason why you can actually do mathematical operations on `char`s in C, such as adding two letters (which usually makes no sense of course). For the future, if we talk about the *mathematical*, or the *`int`-value* of a char, we mean the real numerical value that is stored (in this case `97`), and if we talk about the *character* or *`char`* value, we refer to the letter that is represented by the char (in this case '`a`'). (And by letter we mean all signs in the ASCII-table, including special characters and numbers).

Example: We can assign the *mathematical* 0 to a *char*, or we can assign the character '0' (note the quotes) to this `char`, which would then be translated to the ASCII-value 48 and stored. This is a huge difference and very important to understand. In the special case of zero, an additional way to store the *mathematical* 0 has been introduced: The NULL-character '\0'. You can use this character equivalent to the *mathematical* 0.

In the tutorials and in the last homework you already should have learned about strings. But what is the connection between an array of `char`s and a string? A string is an array of `char`s, where we don't consider the `char`s as single, independent values or letters (just as

e.g.in an `int`-array) but as related letters of a single word. This means basically that we call a `char`-array a string, if we treat the content as text, and not as single `char`s.

In this homework, you will create three functions to process and analyze those strings.

Tool Box

The explanations below **may** help you to solve this problem.

TERMINATING NULL

When a `char`-array is used to store strings, it is also often called "buffer". Every buffer has a fixed size, which means that most of the time the string in the buffer will be shorter than the buffer itself. But when we are processing the buffer (e.g. printing, converting etc.) it is indispensable to know where the valid information (the string) ends. This is why the terminating NULL-character is of very high importance. As the name already indicates, this NULL-character terminates the string and is therefore ALWAYS the subsequent element of the last valid letter of our string. It follows that a buffer of size ten can only contain a maximum of nine letters, as the last one has to be a NULL-character. As explained above, the NULL-character can be expressed as plain 0 (without quotes) or as '\0'. Both variants result in a *mathematical* 0 (All bits of this `char` are 0). **If not stated otherwise, you can always assume that the strings you are receiving are correctly NULL-terminated, and the strings you are producing/returning have to be correctly NULL-terminated as well!** If you do not use the `char`-array as buffer, but to store single, independent values, you don't need the terminating NULL.

ARRAYS AS FUNCTION PARAMETERS

In the last tutorials, you should have learned about local variables. For example, if you are in the main function and call another function with an `int`-value as parameter, the value will be a local variable in that function. Changes to this local variable in the function will not affect the value of our `int`-variable in the main, as we only **passed the value** of our `int`, and a local copy of it has been created, changed, and at the end discarded in the other function. In contrast to that, when we pass an array to a function, we **pass it by reference**, which basically means that we passed the address of the array. In future tutorials, the concept of these passes by reference will be discussed in detail. For now, it is only important for you to know that you can treat the passed arrays just as you were in the main function. Changes done to the arrays will also affect the original array as you are working directly at the respective piece of memory and not at a local copy of the array. It is very important that you can clearly differentiate this from the normal passed-by-value cases.

ASCII-CODES

As described above, every character has a specific ASCII Code. This homework will be a lot easier if you make use of these codes. You can find them online.

Specifications

When the function `to_lower(char array[])` is called by a main function, the following should happen:

- The string in `array` is converted to completely lower case, which means that all higher case letters in the string are converted to lower case
- You may assume that no umlauts (ä,ö,ü) will appear in input
- Other characters (lower case letters, numbers…) remain unchanged

When the function `detect(char array[], char code[])` is called by a main function, the following should happen:

- It is tested, whether the string in `array` contains the string in `code`. E.g. if the string in `code` is "`sos`", it is tested if the string in `array` contains "`sos`" (case sensitive!). By "contains", we mean that all letters in `code` have to appear in `array` in the correct order and right after each other. (e.g. "`Es`**`sos`**" contains "`sos`", but "**`so`**`n`**`s`**" and "`b`**`oss`**" do not)
- If `array` contains `code`, `1` is returned, otherwise `0` is returned

When the function `clean(char array[], char spam)` is called by a main function, the following should happen:

- The string in `array` is "cleaned" from the character in `spam`, which means that all appearances of `spam` in the string are deleted
- The remaining characters are properly re-placed in the array (no spaces where the `spam`s have been before)
- Example: `array: Hello  spam: l` → `array: Heo`

Tasks

We provide you the template file ***string_handling.c*** on Moodle. Complete the functions `to_lower()`, `detect()` and `clean()`, so that the program works as specified above. Please note:

- You are only allowed to write code in the spots indicated in the functions. Do not change anything else. Also do not include any other libraries than the ones already included.
- Do not print anything
- You are provided with an additional C-File ***main_string.c***. This file contains a main-Function, which you can use to test your functions. Store the two C-Files in the same directory on your computer, and compile and run the main file:

  ```
  gcc -Wall -Werror -o test main_string.c
  ```

  The main will then ask you which of the functions you want to test, ask you for the respective input, and finally print the results of your calculations. You can then check by hand if they are correct. (Note: You don't need to take care of the stored "ENTER" from `fgets()`, this is done automatically)
- When you are finished, put ***string_handling.c*** directly in a zip-File named "HW3_Mustermann_Max" (with your name of course) and upload this file to Moodle. Do NOT upload your main-File and do NOT put any folders into the zip-File.

Threshold
To pass this homework, your program has to fulfill the following conditions:

- Your file must be compilable and runnable without any errors and with the flags -Wall -Werror enabled and fulfil the above defined functionality
- Each of your three functions will be tested with 10 different cases, which totals to 30 testcases. To pass this assignment, you have to succeed in at least 80% (=24) of the test cases

Information on Plagiarism and Copyright

As already announced in the lecture and the tutorials, you are supposed to do this homework alone. This homework is an official part of the exam at the end of this semester, and therefore fraud or plagiarism will be pursued and punished.

You are encouraged to discuss **specific** problems with other students (like e.g. “I don’t know how to print a string, can you help me?”), but you are forbidden to share ideas on the approach/structure of your functions, and of course to show other students your solution. This will result in a similar control-flow and structure of your submissions, which we will be able to detect even if you change variable names or loops etc.

We will compare your submissions with a code-optimized plagiarism scanner and if we find plagiarized code, you will fail this submission. If you are found guilty of plagiarism twice, you will lose the homework bonus, and in severe cases we will take further steps.

This homework is intellectual property of the TUM, namely the Chair of Image-Based Biomedical Computing. Publishing either this homework task itself or your solutions to it in any form (including on the MSE-Cloud) is strictly forbidden and can be pursued TUM-internally as well as legally.