CSE 220: Systems Fundamentals I

Homework #3

Spring 2017

Assignment Due: March 10th, 2017 by 11:59 pm

Assignment Overview

In this assignment you will be creating functions that process strings (1D arrays of characters). These are the kinds of functions one might find in a string library/API. The theme of the assignment is basic array manipulation and will give you additional experience working with strings in MIPS.

You **MUST** implement all of the functions in the assignment as defined. It is OK to implement additional helper functions of your own in `hw3.asm`.

⚠ **You MUST follow the MIPS calling and register conventions. If you do not, you WILL lose points.**

⚠ Do not submit a file with the functions/labels `main` or `_start` defined. You will obtain a ZERO for the assignment if you do this.

ℹ If you are having difficulties implementing these functions, write out the pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic to MIPS.

ℹ When writing your program, try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly.

Getting Started

From the Resources section of Piazza download the files `hw3.asm` and `hw3_main.asm`, which you need for the assignment. At the top of your `hw3.asm` program in comments put your name and SBU ID number.

```
# Homework #3
# name: MY_NAME
# sbuid: MY_SBU_ID
```

How to test your functions

To test your functions, simply open the provided `hw3_main.asm` file in MARS. Next, assemble the `hw3_main` file and run. MARS will take the contents of the file referenced with the `.include` at the end of the file and add the contents of your `hw3.asm` file to the main file before assembling it. Once the contents have been substituted into the file, MARS will then assemble it as normal.

Each of the tests in the main file calls the functions you are to implement with one of the sample test cases. You should modify these files or create your own files to test your functions with more test cases.

⚠ Your assignment will not be graded using these tests!

The `hw3_main.asm` file will not be graded. You will only submit your `hw3.asm` file via Sparky. Make sure that all code required for implementing your functions (`.text` and `.data`) are included in the `hw3.asm` file! To make sure that your code is self-contained, try assembling your `hw3.asm` file by itself in MARS. If you get any errors (such as a missing label), this means that you need to refactor (reorganize) your code, possibly by moving labels you inadvertently defined in `hw3_main.asm` to `hw3.asm`.

⚠ It is highly advised to write your own main programs (new individual files) to test each of your functions thoroughly.

⚠ Make sure to initialize all of your values within your functions! Never assume registers or memory will hold any particular values!

Functions to Implement

All functions implemented in the assignment must be placed in `hw3.asm` and follow the standard MIPS register conventions for functions taught in lecture. If you write your own helper functions, these also must be included in `hw3.asm`.

⚠ All functions are CASE-SENSITIVE. This means that `'T'` DOES NOT match `'t'`.

a. `int indexOf(char[] str, char ch, int startIndex)`

   This function searches through a null-terminated string for a particular character, starting at a given index. The function makes no changes to `str` in memory. The notation `char[]` indicates that the function is expecting the starting address of a string, i.e., the address of the first character of the string. The address is not necessarily word-aligned.

   - `str`: The null-terminated string to search through.
   - `ch`: The ASCII character to search for. The ASCII value of the character is passed, not the address of the character.
   - `startIndex`: The index to start the search from.
   - *returns*: The index of the character `ch` in `str` if it was found, or -1 if the character was not

found or if `startIndex` is negative.

Examples:

Code	Return Value
`indexOf("abracadabra", 'b', 0)`	1
`indexOf("abracadabra", 'b', 3)`	8
`indexOf("abracadabra", 'b', 8)`	8
`indexOf("abracadabra", 'c', 6)`	-1
`indexOf("Stony Brook Univ", 't', -2)`	-1
`indexOf("", 'q', 0)`	-1

For the second example above, the return value is 8 because the first `'b'` that appears after (or at) index 3 is found at index 8 in the input string.

b. `(char[], int) replaceAllChar(char[] str, char[] pattern, char replacement)`

This function replaces all characters in `str` that matches one of the characters in the null-terminated string `pattern` with the character `replacement`. The function performs an *in-place* substitution, meaning that the original parameter `str` is modified and no additional memory is required. The function makes no changes to `pattern` stored in memory.

Note: The notation `(char[], int)` indicates that the function returns two values: the starting address of a string in `$v0` and a 32-bit integer in `$v1`.

- `str`: The string that will be modified as described above. If `str` is empty (i.e., contains only a null-terminator), the function returns (`str`, -1) and makes no changes to `str`.
- `pattern`: Array of characters to search for. If `pattern` is empty (i.e., contains only a null-terminator), the function returns (`str`, -1) and makes no changes to `str`.
- `replacement`: ASCII character that replaces each instance of the characters from `pattern`. The ASCII value of the character is passed, not the address of the character.
- *returns* in `$v0`: The starting address of `str`.
- *returns* in `$v1`: Number of replacements performed, or -1 if error.

Examples:

Code	`str` After Call	Return Values
`replaceAllChar("Stony Brook", "oBhy", 'q')`	`"Stqnq qrqqk"`	address of `str`, 5
`replaceAllChar("Stony Brook", "sO7h", 'S')`	`"Stony Brook"`	address of `str`, 0
`replaceAllChar("Stony Brook", "", 'S')`	`"Stony Brook"`	address of `str`, -1
`replaceAllChar("", "heN2", 'h')`	`""`	address of `str`, -1
`replaceAllChar("  ", "8 0", 'Z')` (`str` is 3 spaces)	`"ZZZ"`	address of `str`, 3

c. `int countOccurrences(char[] str, char[] searchChars)`

This function searches through `str`, looking for characters that appear in `searchChars`. The function returns the number of times the characters from `searchChars` appear in `str`. Your function may assume that no character appears more than once in `searchChars`. If `str` and/or `searchChars` is an empty string, the function returns 0. The functino makes no changes to `str`.

- `str`: The null-terminated string to search.
- `searchChars`: The null-terminated string of characters to search for.
- *returns*: The number of times characters from `searchChars` appear in `str`.

Examples:

Code	Return Value
`countOccurrences("Let's Go Seawolves!", "qsgo!")`	5
`countOccurrences("Winter Wonderland", "uwE3?y")`	0
`countOccurrences("", "h6sw")`	0
`countOccurrences("New York City", "")`	0

d. `(char[], int) replaceAllSubstr(char[] dst, int dstLen, char[] str, char[] searchChars, char[] replaceStr)`

This function replaces every instance of each character in `searchChars` found in `str` with `replaceStr` and stores the modified, null-terminated string in `dst`, leaving `str` unchanged.

`replaceStr` is not necessarily a single character in length. Therefore, the function must verify the modified string length with the null-terminator is $<=$ `dstLen`. If it is larger, then replaceAllSubstr returns error and makes no changes to `dst`.

Assume that `str`, `searchChars` and `replaceStr` are null-terminated. The function makes no changes to `str`, `searchChars` or `replaceStr`. The function makes no changes to `dst` except what is required to satisfy the function specification.

- `dst`: Address of character array to store the new string. The function must null-terminate this string.
- `dstLen`: Number of bytes in the `dst` array. If `dst` is not of sufficient length to store the modified string (including the null-terminator), the function returns (`dst`, -1) and makes no changes to `dst`.
- `str`: The input string. If `str` is empty (i.e., contains only a null-terminator), the function returns (`dst`, -1) and makes no changes to `dst`.
- `searchChars`: The string to search for inside of `str`. If `searchChars` is empty (i.e., contains only a null-terminator), the function returns (`dst`, -1) and makes no changes to `dst`.

- `replaceStr`: The string to replace with. If `replaceStr` is empty, the function simply deletes all instances of `searchChars` when writing `dst`.

  ℹ Note that the address of this string is the fifth function argument and is therefore placed on the top of the runtime stack by the caller function.

- *returns* in `$v0`: The starting address of `dst`.
- *returns* in `$v1`: Number of replacements performed, or -1 if error.

❗ `replaceAllSubstr` MUST call `countOccurrences`.

⚠ In the examples below, the `dst` array is filled with garbage to emphasize the fact that **any and all** of your functions might be tested with garbage-filled memory. NEVER assume that a particular value (like 0) is stored in any register or any memory cell.

Examples:

Code	
`replaceAllSubstr("*****?????ggggg123456789", 24, "Seawolves Rule!", "oTs e", "XY")`	
`dst` **After Call**	**Return Values**
`"SXYawXYlvXYXYXYRulXY!\089"`	address of `dst`, 6

Code	
`replaceAllSubstr("whatsupwithyou", 14, "dingbat", "GrPQ", "Ugh")`	
`dst` **After Call**	**Return Values**
`"dingbat\0ithyou"`	address of `dst`, 0

Code	
`replaceAllSubstr("hocus, pocus alimagocus@", 24, "Umadgic Bbun#ny", "UdB#", "")`	
`dst` **After Call**	**Return Values**
`"magic bunny\0 alimagocus@"`	address of `dst`, 4

Code	
`replaceAllSubstr("jei8sakwhrdwK", 13, "curious", "icoz9", "Jnn")`	
`dst` **After Call**	**Return Values**
`"jei8sakwhrdwK"`	address of `dst`, -1

Code	
`replaceAllSubstr("abracadabradoodle", 17, "", "h4", "Q")`	
`dst` **After Call**	**Return Values**
`"abracadabradoodle"`	address of `dst`, -1

Code	
`replaceAllSubstr("??!?@***Hwdkfhjwe", 17, "MIPS is Awesome!", "", "A")`	
`dst` **After Call**	**Return Values**
`"??!?@***Hwdkfhjwe"`	address of `dst`, -1

e. `(int, int) split(int[] dst, int dstLen, char[] str, char delimiter)`

This function tokenizes (splits) a string into its constituent substrings as delimited by the `delimiter` character. For example, suppose we have a string "Stony Brook University" and the space character is the delimiter. Then we would have three tokens: "Stony", "Brook" and "University". Note that the delimiters are not included in the tokens.

The starting address of each token found in `str` is written into `dst` sequentially. Thus, the function builds an array of addresses, namely, the starting address of each token in `str` that is found by the function. The function replaces each instance of `delimiter` in `str` with a null-terminator. The function makes no changes to `dst` except what is required to satisfy the function specification.

`dstLen` indicates how many 32-bit words are in the `dst` array. `dstLen` therefore indicates the maximum number of memory addresses that can be written to `dst`. If `str` contains more than `dstLen` tokens, the function writes as many addresses as it can to `dst` before running out of room in the array. The function should only tokenize (change the delimiter to '\0') if `dst` has room to hold the token.

- `dst`: Address of the word-aligned array that stores the starting addresses of each substring.
- `dstLen`: Number of 32-bit words in the `dst` array. `dstLen` is guaranteed to be at least 1.
- `str`: The null-terminated string to split into tokens.
- `delimiter`: ASCII character to serve as the delimiter. The ASCII value of the character is passed, not the address of the character.
- *returns* in `$v0`: The number of addresses in `dst` if `delimiter` is valid, or -1, otherwise.
- *returns* in `$v1`: 0 if the `str` was completely tokenized. Returns -1 if not every token's address could be written to the `dst` array (due to lack of space). Also returns -1 if `str` is empty.

Special cases the function is able to handle:

- The `delimiter` is not found in `str`. In this case, the entire string `str` is treated as a single token. This is considered a successful tokenizing.
- The first character in `str` is a delimiter. In this case, the first token will be an empty string. The second token will start at index 1 of `str` (because the delimiter is at index 0).
- The last character in `str` before the null-terminator is a delimiter. In this case, the last token will be an empty string and "starts" (and "ends") at the address of the null-terminator.
- Exactly two delimiters are next to each other somewhere in the middle of the string, with at least one non-delimiter or non-null-terminator character on both sides of the two delimiters. In this case, there is a token in between them, namely, an empty string. The address of this token is given by the address of the second delimiter.

Special cases the function does not need to handle (and will not be checked for during grading):

- More than two delimiters appear next to each other contiguously in the string `str`.
- The string `str` starts with two delimiters.
- The string `str` ends with two delimiters.
- The string `str` contains only one character and that character is a delimiter.

❗ `split` MUST call `indexOf`.

Examples:

Values stored in `str` after function call were added to all examples.
Invalid Token example removed.

Assume that string `str` starts at address `0x41` in the following examples.

General case of clean input:

Code	Return Values
`split(dst_addr, 12, "Happy birthday", 'a')`	3, 0
`dst` **After Call**	
`[0x41, 0x43, 0x4E, ... (remaining contents unchanged)]`	
`str` **After Call**	
`"H\0ppy birthd\0y"`	

Not enough space in `dst` to store all the tokens' starting addresses:

Code	Return Values
`split(dst_addr, 3, "hokus pokus smokus!", 'k')`	3, -1
`dst` **After Call**	
`[0x41, 0x44, 0x4A]`	
`str` **After Call**	
`"ho\0us po\0us smokus!"`	

Delimiter not found in `str`. Treat string `str` as one token:

Code	Return Values
`split(dst_addr, 2, "Let's go Seawolves!", '$')`	1, 0
`dst` **After Call**	
`[0x41, ... (remaining contents unchanged)]`	
`str` **After Call**	
`(contents unchanged)`	

A single delimiter encountered at start of `str`:

Code	Return Values
`split(dst_addr, 10, "Xstony XbroXok", 'X')`	4, 0
`dst` **After Call**	
`[0x41, 0x42, 0x49, 0x4D, ... (remaining contents unchanged)]`	
`str` **After Call**	
`"\0stony \0bro\0ok"`	

A single delimiter encountered at end of `str`:

Code	Return Values
`split(dst_addr, 8, "Computer Science", 'e')`	4, 0
`dst` **After Call**	
`[0x41, 0x48, 0x4E, 0x51, ... (remaining contents unchanged)]`	
`str` **After Call**	
`"Comput\0r Sci\nc\0"`	

Two delimiters are next to each other in `str`:

Code	Return Values
`split(dst_addr, 8, "ugga mugga?", 'g')`	5, 0
`dst` **After Call**	
`[0x41, 0x43, 0x44, 0x49, 0x4A, ... (remaining contents unchanged)]`	
`str` **After Call**	
`"u\0\0a mu\0\0a?"`	

Hand-in Instructions

See Sparky Submission Instructions on Piazza for hand-in instructions.

There is no tolerance for homework submission via email. Work must be submitted through Sparky. Please do not wait until the last minute to submit your homework. If you are struggling, stop by office hours for additional help.