Assignment 6: String Library

In this assignment, you'll create your own library of string functions. You'll have the opportunity to practice manipulating strings and managing memory. Additionally, you'll learn the role of header and library files.

NOTE: You may **not** call functions in string.h but you can use other code in the Standard C Library.

Implement the following functions. Be sure that any string that you create or modify is in fact a string, i.e., an array of char (whether represented as a pointer or as an arrray), terminated with the null character, '\0'.

Additionally, you should write a driver (i.e. a C program that includes and then uses your library functions), which tests each of your functions on real data.

- **[3 points]** int str_len(char *s)

  Returns the number of characters in the string s (up to but not including the '\0' character).

  - **Note**: Use pointer notation, not array notation for this. An example strlen implementation is shown below. Before just using this code, please understand what it is doing (you will be asked to explain why that code works). There are multiple ways you could implement this, so if your implementation is different from the below example, that is not a problem (as long as it works correctly of course).

- **[5 points]** int all_letters(char *s)

  Returns 1 if all of the characters in the string are either upper-case or lower-case letters of the alphabet. It returns 0 otherwise.

- **[5 points]** num_in_range(char *s1, char b, char t)

  returns the number of characters c in s1 such that b<=c<=t

- **[8 points]** diff(char *s1, char *s2)

  returns the number of positions in which s1 and s2 differ, i.e., it returns the number of changes that would need to be made in order to transform s1 into s2, where a change could be a character substitution, an insertion, or a deletion.

- **[5 points]** void shorten(char *s, int new_len)

  Shortens the string s to new_len. If the original length of s is less than or equal to new_len, s is unchanged

- **[5 points]** int len_diff(char *s1, char *s2)

  Returns the length of s1 - the length of s2

- **[5 points]** void rm_left_space(char *s)

  removes whitespace characters from the beginning of s

- **[5 points]** void rm_right_space(char *s)

  removes whitespace characters from the end of s

- **[5 points]** void rm_space(char *s)

  removes whitespace characters from the beginning and the ending s

- **[10 points]** int find(char *h, char *n)

  returns the index of the first occurence of n in the string h or -1 if it isn't found.

- **[10 points]** char *ptr_to(char *h, char *n)

  returns a pointer to the first occurence of n in the string h or NULL if it isn't found

- **[8 points]** is_empty(char *s)

  returns 1 if s is NULL, consists of only the null character ('') or only whitespace. returns 0 otherwise.

- **[10 points]** str_zip(char *s1, char *s2)

  Returns a new string consisting of all of the characters of s1 and s2 interleaved with each other. For example, if s1 is "Spongebob" and s2 is "Patrick", the function returns the string "SPpaotnrgiecbkob"

- **[10 points]** void capitalize(char *s)

  Changes s so that the first letter of every word is in upper case and each additional letter is in lower case.

- **[10 points]** int strcmp_ign_case(char *s1, char *s2)

  Compares s1 and s2 ignoring case. Returns a positive number if s1 would appear after s2 in the dictionary, a negative number if it would appear before s2, or 0 if the two are equal.

- **[6 points]** void take_last(char *s, int n)

  Modifies s so that it consists of only its last n characters. If n is ≥ the length of s, the original string is unmodified. For example if we call take_last("Brubeck" 5), when the function finishes, the original string becomes "ubeck"

- **[10 points]** dedup(char *s)

returns a new string based on s, but without any duplicate characters. For example, if s is the string, "There's always money in the banana stand.", the function returns the string "Ther's alwymonitbd.". It is up to the caller to free the memory allocated by the function.

- **[10 points]** pad(char *s, int d)

  returns a new string consisting of all of the letters of s, but padded with spaces at the end so that the total length of the returned string is an even multiple of d. If the length of s is already an even multiple of d, the function returns a copy of s. The function returns NULL on failure or if s is NULL. Otherwise, it returns the new string. It is up to the caller to free any memory allocated by the function.

- **[10 points]** ends_with_ignore_case(char *s, char *suff)

  returns 1 if suff is a suffix of s ignoring case or 0 otherwise.

- **[10 points]** char *repeat(char *s, int x, char sep)

  Returns a new string consisting of the characters in s repeated x times, with the character sep in between. For example, if s is the string all right, x is 3, and sep is , the function returns the new string all right,all right,all right. If s is NULL, the function returns NULL. It is up to the caller to free any memory allocated by the function.

- **[10 points]** char *replace(char *s, char *pat, char *rep)

  Returns a copy of the string s, but with each instance of pat replaced with rep, note that len(pat) can be less than, greater than, or equal to len(rep). The function allocates memory for the resulting string, and it is up to the caller to free it. For example, if we call replace("chicken X", "X", "nugget"), what is returned is the new string "chicken nugget" (but remember, pat could be longer than an individual character and could occur multiple times).

- **[10 points]** char *str_connect(char **strs, int n, char c)

  Returns a string consisting of the first n strings in strs with the character c used as a separator. For example, if strs contains the strings {"Washington", "Adams", "Jefferson"} and c is '+', the function returns the string "Washington+Adams+Jefferson"

- **[10 points]** void rm_empties(char **words)

  words is an array of string terminated with a NULL pointer. The function removes any empty strings (i.e., strings of length 0) from the array.

- **[10 points]** char **str_chop_all(char *s, char c)

  Returns an array of string consisting of the characters in s split into tokens based on the delimeter c, followed by a NULL pointer. For example, if s is "I am ready for a nice vacation" and c is ' ', it returns {"I", "am", "ready", "for", "a", "nice", "vacation", NULL}

Arch

Remember that we keep our function declarations in a header file, a .h file. Each of your functions should be in a separate .c file (e.g., you'll have all_letters in a file called all_letters.c file, the num_in_range in a file called num_in_range.c .

So you'll have:

- A collection of .c files:
  - one for each function in the library
  - a test program
- A single .h file, which includes the declarations for each of the functions in the library. This is #included by each of the .c files.

Generating the Library

To create the library file, we use the ar command. The syntax is:

```
ar rcs NameOfTheLibraryFileToCreate listOfFilesToIncludeInTheLibrary
```

Don't forget that you're including the binary files, i.e., the .o files, not your c source files.

So suppose you have all of your function .o files in a single directory, and you'd like to call your library file libstr2107.a, you'd type:

```
ar rcs libstr2107.a *.o
```

To double-check, you can try:

```
ar t libstr2107.a
```

if you see a list of your .o files, you've done it right. The library file name should begin lib and should have a .a extension.

Using the library with your driver (10 points)

Write a very simple, basic program to test your functions. Suppose that it's in a file called strtester.c. To compile it, using your new string library, libstr2107.a, type:

```
gcc -o strtester strtester.c -LdirectoryWhereYouPutTheLibrary -lstr2107
```

so, if libstr2107.a is in your current directory, you'd type:

```
gcc -o strtester strtester.c -L. -lstr2107
```

Note that at the end of the line, it's -l (lower case l, not the number 1), and it's just str2107 not libstr2107.a.

The meaning of the command line is:

- **-o strtester** name the output file strtester

- **-L.** look for library files in the current directory
- **-lstr2107** link the contents of the file libstr2107.a

Deliverables

Please submit either a single tar file or zip file containing all of your .c files, your .h file and your string library file (your .a file). In order to help the TA keep track of everyone's files, please include your name in the name of the tar/zip file.

Some Tips, Reminders, etc.

Pointer vs Array Notation

Though it's not a formal requirement, it is suggested that you try to do some of these using pointer notation instead of array notation.

For example, we could write a string length function as:

```
int strlen(char s[])
{
   int i=0;
   while (s[i]!='\0')
      i++;
   return i;
}
```

We could also write:

```
int strlen(char *s)
{
   char *t=s;

   while (*t!='\0')
```

```
      t++;
   return t-s;
}
```

gdb

We spent time with GDB for a reason. Use it.

stack memory vs. heap memory

All of this about *"The caller is responsible for freeing the memory ..."* should tell you that inside the function you're going to be allocating the memory for what's returned. Remember to use malloc (or calloc) and don't return a pointer to local stack memory. For example, something like this:

```
char *func()
{
   char str[50];

   ...

   return str;
}
```

might compile, but it's wrong. str is allocated on func's stack, and the memory used by str will be likely be given to some other function as the program continues to run. Instead, you'd do something like:

```
char *func()
{
   char *str;

   ...
```

```
   if ((str=malloc(50))==NULL) /* allocate 50 bytes and check
*/
      return NULL;              * to see if they were
successfully */
                                * allocated */

   ...
   return str;
}
```

As we've discussed in class, this is something that you didn't have to worry about in Java, because Java arrays are allocated on the heap. Remember that the equivalent Java would be:

```
char[] func()
{
  char str[] = new str[50];

  ...

  return str;
}
```

and as we also said in class, Java's new operator serves the same purpose as C's malloc( )

string literals

Here's another thing that you didn't need to worry about in Java that might cause some headaches in C. Remember that these two declarations are not exactly the same:

```
char *str01 = "What time does class end?";
```

```
char str02[] = "What time does class end?";
```

They both look the same when you print them, but if you try to modify str01, you'll get in trouble. The memory that str01 points to is read-only. str02 isn't, but it's only as much space as is required to hold the letters 'W', 'h', 'a', ..., and the null character ('\0').

None of this is likely to be a big deal in any of the functions you write, but it might come up when you're trying to test a function. For example, something like:

```
char *str01 = "     Where's the remote control?     ";

...

strip(str01);
```

will give you a segmentation fault, but this doesn't necessarily mean that there's a problem in the strip function.

For more information, please take a look at K&R section 5.5.