Laboratory Exercise

Text string related functions

Purpose of the Exercise

This exercise will help you do the following:

- Use dynamic allocation/deallocation.
- Practice working with null-terminated strings.
- Demonstrate competencies in implementing functions similar to the ones from <string.h>.

Overview

<string.h>

The header <string.h> contains many useful functions for processing null-terminated strings of text, e.g. `strcpy()`, or non-null-terminated blocks of binary data like `memcpy()`. As most of these functions consist of a simple loop over elements in a buffer of characters or data, mastering use and implementation of functions like these is an important component in building a deeper understanding of their underlying concepts: iteration statements, arrays, pointers and pointer arithmetic. This is the purpose of this exercise.

This laboratory exercise consists of two steps.

In the first step, inside *q.c* you will define 4 functions missing from the provided driver program to ensure that it generates the correct output. Their declarations have been provided in a header file *q.h*. During compilation at this step you will add a pre-processor symbol `USE_STRING` so that the program relies on functions from *<string.h>*.

In the second step, inside *my_string.c* you will implement required functions from this header yourself and compile the code without the pre-processor symbol so that the program still generates the same output. Declarations of required string functions have been provided to you in a header file *my_string.h* alongside web links to the reference documentation that offers more information about behavior of each function.

Reserved Identifiers

As you have learnt earlier in this course, all *valid* identifiers in the C code, such as names of variables or functions, must start with an underscore or a letter, and may include a digit. They also cannot be any of the keywords defined by the standard of the language specification.

You may also remember that there are some additional rules about *reserved* identifiers. Such identifiers may be *valid*, but should not be used as they may result in *undefined behavior*. Particularly, all external identifiers used by the Standard Library are *reserved*; it is still legal to declare a variable of a matching name in a block scope or as a function parameter, as it implies *no linkage*, or a function of a matching name with `static` storage, as it implies *internal linkage*. The Standard Library header that declares a conflicting symbol must not be included in such a translation unit.

However, such identifiers used with *external linkage* lead to an undefined behavior.

In this exercise, you will both use the Standard Library functions and implement your own. To avoid name conflicts, all your functions will use a prefix `my_`, for example, `my_strlen()`. The code uses macros like `STRLEN` that will resolve either to `strlen` or `my_strlen` depending on whether another macro `USE_STRING` was specified from the compilation command line.

Constraints

While implementing this exercise, you must follow these constraints:

- You must not call `malloc()`/`calloc()`/`realloc()` or `free()` functions directly; for any memory allocation use `debug_malloc()` and `debug_free()` provided by the driver.
- You must not call any function from *<string.h>* directly; you must use macros defined in *q.h*.
- You must observe `const`-ness and `const`-correctness.
- The file *q.c* must only include *q.h*. The file *my_string.c* must only include *my_string.h*.
- You must ensure there are no memory leaks - all memory must be properly released.
- You must ensure a buffer overrun never happens which is where you perform any access operation on data out of bounds of a buffer. In this exercise, it may happen if you forget that a string of text includes a null-terminator character.

Tasks

In this exercise you will implement functions of a program that perform testing of strings of text representing directory paths.

In a header file *q.h* you can find declarations of the following functions:

- `build_path()`

  The function takes in a path to a parent folder, a path separator sequence (for Linux paths it is `"/"`, for Windows paths it is `"\\"`), and an array of subdirectories with its element count. It combines the parent folder and the subdirectories into a single path using the separator.

  ```
  const char* build_path(
      const char* parent,
      const char* separator,
      const char* const folders[],
      size_t count
  );
  ```

  Your implementation must demonstrate calls to `STRCPY` and `STRCAT` (see macros in *q.h*). You can use other macros as you find suitable.

  The returned path will be later deallocated by the driver with a call to `debug_free()`.

- `compare_string()`

  The function prints out a statement about a 3-way comparison of two strings.

  ```
  void compare_string(
      const char* lhs,
      const char* rhs
  );
  ```

  Your implementation must demonstrate calls to `STRCMP` (see macros in *q.h*). You can use other macros as you find suitable.

- `describe_string()`

The function prints out the length of a provided string of text.

```
void describe_string(const char* text);
```

Your implementation must demonstrate calls to `STRLEN` (see macros in *q.h*). You can use other macros as you find suitable.

- `find_string()`

  The function prints out a statement describing a result of searching a string of text (`substring`) within another string of text (`string`).

  ```
  void find_string(
      const char* string,
      const char* substring
  );
  ```

  Your implementation must demonstrate calls to `STRSTR` (see macros in *q.h*). You can use other macros as you find suitable.

Define all these functions in *q.c* to complete the first part of the program.

Then, in the second part you have to implement the string-related functions. In a header file *my_string.h* you can find declarations of the following functions:

-
  ```
  size_t my_strlen(const char* str);
  ```

  Returns a length of the string as a number of characters before the null-terminator.

-
  ```
  char* my_strcpy(char* dest, const char* src);
  ```

  Copies a source string of text (including its null-terminator) into a destination buffer and returns a pointer to that buffer.

-
  ```
  char* my_strcat(char* dest, const char* src);
  ```

  Concatenates (appends) a source string of text (including its null-terminator) at the end of a destination buffer and returns a pointer to that buffer.

-
  ```
  int my_strcmp(const char* lhs, const char* rhs);
  ```

  Compares two strings of text and returns `0` if they are equal, a negative number if the first string should be alphabetically sorted first, or a positive number if the second string should be alphabetically sorted first.

-
  ```
  char* my_strstr(const char* str, const char* substr);
  ```

  Searches for a string of text in another string of text and returns a pointer to the beginning of its first occurrence, or `NULL` if it cannot be found.

Define all these functions in *my_string.c* to complete the second part of the program, and thus the entire program.

Step 1. Prepare your environment

Open your WSL Linux environment, prepare an empty *sandbox* directory where the development will take place, save the provided text files into this directory. Then create new files *q.c* and *my_string.c* and open them for editing.

Step 2. Review the expected output file

Open the output text file and make sure that you know which of the functions was used to generate its parts and that you know how to format the output.

Step 3. Part 1 - *q.c*

Step 3.1. Add file-level documentation

Add the file-level documentation to *q.c*. Remember that every source code file you submit for grading must start with an updated file-level documentation header.

Step 3.2. Implement functions

Provide stub definitions for all required functions declared in *q.h*; for now they do not have to perform the roles. In each stub implementation leave a comment `// TODO` to make sure that you remember to return to this incomplete implementation.

Try to compile this code:

```
gcc -DUSE_STRING -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic -std=c11 -o main main.c q.c
```

Take note that it includes a definition `-D` of a macro `USE_STRING` that in the program enables use of the Standard Library's <*string.h*> header.

Work on the code until it compiles. From there define each function one by one, each time running this command, testing the resulting executable and comparing its output with the provided expected output file:

```
./main > actual-output.txt
diff actual-output.txt expected-output.txt --strip-trailing-cr
```

Edit the code until it compiles and the actual output matches exactly the expected output.

Step 3.3. Add function-level documentation

Add the function-level documentation to every function in *q.c*. Remember that every function in every source code file that you submit for grading must be preceded by a function-level documentation header that explains its purpose, inputs, outputs, side effects, and considerations.

Step 4. Part 2 - *my_string.c*

Step 4.1. Add file-level documentation

Add the file-level documentation to *my_string.c*.

Step 4.2. Implement functions

Provide stub definitions for all required functions declared in *my_string.h*; for now they do not have to perform the roles. In each stub implementation leave a comment `// TODO` to make sure that you remember to return to this incomplete implementation.

Try to compile this code:

```
gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic -std=c11
-o main main.c q.c my_string.c
```

Take note that it does **not** include a definition of a macro `USE_STRING` and thus it switches the program to your own implementation of string functions from the *"my_string.h"* header.

Work on the code until it compiles. From there define each function one by one, each time running this command, testing the resulting executable and comparing its output with the provided expected output file:

```
./main > actual-output.txt
diff actual-output.txt expected-output.txt --strip-trailing-cr
```

Edit the code until it compiles and the actual output again matches exactly the expected output.

Step 4.3. Add function-level documentation

Add the function-level documentation to every function in *my_string.c*.

Step 5. Clean up the code

Clean up the formatting, make sure it is consistent and easy to read. Break long lines of code; a common guideline is that no line should be longer than 80 characters.

Avoid repetition. If you see that multiple functions perform the same operation, or that certain statements logically represent distinct functionality, consider abstracting them away into separate helper functions; remember to make them `static`.

Also, review the **Constraints** section of this document to assess whether you have followed all the restrictions imposed in this laboratory exercise.

Step 6. Test the code

Once again before submission test the code. Your actual output must exactly match the contents of the expected output. Use the *diff* command to compare the files; no differences in the output should be reported when the code is compiled without any macro definitions provided in the compilation command.

Submission

Once your implementation of *q.c* and *my_string.c* is complete, again ensure that the program works and that it contains updated file-level and function-level documentation comments.

Then upload the files to the laboratory submission page in Moodle. There are 95 lines of expected output and 95 corresponding automated tests cases. To get the maximum grade, make sure that all test cases match.