Tutorial 10 - String Library

Text string related functions

Learning Outcomes

- Thinking of memory management in terms of dynamic allocation/deallocation.
- Practice working with null-terminated strings.
- Demonstrate competencies in implementing functions similar to some declared in `<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 used by the provided driver program to ensure that it generates the correct output. Their declarations are provided in the following code:

```c
#include <stdio.h>

#ifdef USE_STRING

#include <string.h>

#define STRLEN strlen
#define STRCPY strcpy
#define STRCAT strcat
#define STRCMP strcmp
#define STRSTR strstr

#else

#include "my_string.h"

#define STRLEN my_strlen
#define STRCPY my_strcpy
#define STRCAT my_strcat
#define STRCMP my_strcmp
#define STRSTR my_strstr

#endif

```

```c
/****************************************
 * Defined in qdriver.c                 *
 ****************************************/

void* debug_malloc(
    size_t size
);

void debug_free(
    void* ptr
);

/****************************************
 * Define in q.c                        *
 ****************************************/

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

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

void describe_string(
    const char* text
);

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

```

Copy the above to `q.h`. During compilation you will first define the pre-processor symbol `USE_STRING` so that the program relies on functions declared in `<string.h>`.

In the second step, inside `my_string.c` you will implement some functions similar to `<string.h>` functions and compile the code without the pre-processor symbol so that the program uses your string functions and 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 one or more digits. They also cannot be any of the keywords defined by 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 use the Standard Library functions as well as 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` is specified on 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. This is because `debug_malloc()` and `debug_free()` produce text output that is used for checking your code correctness.
- 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.
- Be judicious about what header files to include in a C file. This is because, the more files are included, the more work the preprocessor and compiler have to do. For example, the file `my_string.c` may include `my_string.h`. This is because satisfying the specifications of this lab task may require some functions in `my_string.c` to call other functions in the same file. When they do so, the compiler would need to know the interface of the function that is called in order to compile. As for `q.c`, the functions in it use macros in `q.h`, so `q.h` ought to be included in `q.c`.
- You must ensure there are no memory leaks. You may use `valgrind` to check all memory is properly released. If your code has any memory error, the Moodle auto-grader may produce a vague error message such as an 'unknown' error. If you see this, you should suspect a memory error and check your program for it.
- You must ensure a buffer overrun never happens. A buffer overrun is where you perform any access operation on data out of the bounds of a buffer. For example, this 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 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.

```c
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 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.

  ```c
  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 you find suitable.

- `describe_string`

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

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

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

- `find_string`

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

  ```c
  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 you find suitable.

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

Then, in the second part of the lab task, you have to implement the equivalent of `<string.h>` functions. In a header file `my_string.h` you can find declarations of the following functions:

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

  Returns the length of the string represented by the number of characters before the null-terminator.

-
  ```c
  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.

-
  ```c
  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.

-
  ```c
  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.

-
  ```c
  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.

Step 1. Prepare your environment

Open your WSL Linux environment, prepare an empty `sandbox` directory where the development will take place, and save the provided text files in this directory. Download requested files `q.h`, `q.c`, and `my_string.c` and open them for editing.

Step 2. Review the expected output file

Open the output text file and see which function is to generate each part and observe 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`; as stubs they do not have to do anything yet. 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-errors -std=c11 -o q.out qdriver.c q.c
```

Take note that it includes a definition of a macro `USE_STRING` which is indicated by `-DUSE_STRING`. In the program this 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:

```
./q.out > actual-output.txt
diff -y --strip-trailing-cr --suppress-common-lines actual-output.txt
expected-output.txt
```

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 a start they do not have to do anything. 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-errors -
std=c11 -o q.out qdriver.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 declared in 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:

```
./q.out > actual-output.txt
diff -y --strip-trailing-cr --suppress-common-lines actual-output.txt
expected-output.txt
```

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 and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit files `q.h`, `q.c`, and `my_string.c`.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
   - F grade if your submission doesn't compile with the full suite of `gcc` options.
   - F grade if your submission doesn't link to create an executable file.
   - A proportional grade if the program's output doesn't fully match the correct output.
   - A+ grade if the submission's output fully matches the correct output.
   - Possible deduction of one letter grade for each missing file header documentation block. Every submitted file must have one file-level documentation block and each function must have a function-level documentation block. A teaching assistant may physically read submitted files to ensure that these documentation blocks are authored correctly. Each missing block may result in a deduction of a letter grade. For example, if the automatic grader gave your submission an A+ grade and one documentation block is missing, your grade may be later reduced from A+ to B+. Another example: If the automatic grader gave your submission a C grade and two documentation blocks are missing, your grade may be later reduced from C to F.