Extreme Edge Cases

Entire Assignment due **2019-01-28 23:59**
Graded files:

- camelCaser.c
- camelCaser_tests.c

Content

Learning Objectives

The learning objectives for Extreme Edge Cases are:

- Test Driven Development
- Thinking of Edge Cases
- String Manipulation
- C Programming

Backstory

What makes code good? Is it camelCased strings? Good comments? Descriptive variable names, perhaps?

One thing we know is that good code is generally modular - it consists of discrete "units" of functionality that are only responsible for very specific and certain behavior. In our case, working with C, these "units" are functions.

For example, the C string function `strlen` (https://linux.die.net/man/3/strlen) is solely responsible for determining the length of a string; it doesn't do any I/O or networking, or anything else. A function that knows all and tries to do all would be bad design, and testing whether that kind of function adheres to expectations would be nontrivial.

A programmer might ask, "do my units of work behave the way I expect?" or "if my function expects a string, how does it behave when given NULL?". These are crucial questions, since ensuring that units of code work exactly the way one would expect makes it easy to build reliable and robust software. An unreliable unit in a large system can affect the entire system significantly. Imagine if `strcpy` (http://linux.die.net/man/3/strcpy), for example, did not behave properly on all inputs; all of the higher-level units that use `strcpy` (http://linux.die.net/man/3/strcpy), and all of the units that, in turn, use those units, would in-turn have unpredictable behavior, and so the unreliability would propagate through the whole system.

Enter unit testing.

Unit Testing

Unit testing is a ubiquitous and crucial software development method used heavily in industry. According to artofunittesting.com (http://artofunittesting.com/definition-of-a-unit-test/), "a unit test is an automated piece of code that invokes a *unit of work* in the system *and then checks a single assumption about the behavior of that unit of work*". This sounds like testing - leave it to the QAs, right? Actually, developers, much to their chagrin, are expected to write their own unit tests.

In order to write effective unit tests, all possible cases of input to a unit (mainly functions, in C), including edge cases, should be tested. Good unit tests test (extreme) edge cases, making sure that the discrete unit of functionality performs as specified with unexpected inputs.

In this MP, your goal is to create and test the behavior of an arbitrary string manipulation function to determine if it is reliable, predictable, and correct. While writing your functions, try to write modular code, as this will make your life easier when you test it. You'll learn how to write effective test cases - an incredibly helpful skill for the rest of the course. Finally, you'll be able to take these skills to Facenovel for your next internship and impress your coworkers.

Camel Caser

We have chosen

```
char **camel_caser(const char* input)
```

as your arbitrary string manipulation function.

Your manager at Facenovel, to celebrate Hump Day, has asked all of the interns to implement a brand new camelCaser to convert sentences into camelCase (https://en.wikipedia.org/wiki/Camel_case). To give you a return offer, he also assigned you to write test cases for all the other interns' implementations of camelCaser, with the implementations hidden from you.

Let's say I want to get a sequence of sentences in camelCase. This is the string passed into your method:

```
"The Heisenbug is an incredible creature. Facenovel servers get their power from its inde
terminism. Code smell can be ignored with INCREDIBLE use of air freshener. God objects ar
e the new religion."
```

Your method should return the following:

```
["theHeisenbugIsAnIncredibleCreature",
"facenovelServersGetTheirPowerFromItsIndeterminism",
"codeSmellCanBeIgnoredWithIncredibleUseOfAirFreshener",
"godObjectsAreTheNewReligion",
NULL]
```

The brackets denote that the above is an array of those strings.

Here is a formal description of how your camelCaser should behave:

- You can't camelCase a NULL pointer, so if `input` is a NULL pointer, return a NULL pointer.
- If `input` is NOT NULL, then it is a NULL-terminated array of characters (a standard C string).
- A input sentence, `input_s`, is defined as any *MAXIMAL* substring of the input string that ends with a punctuation mark. This means that all strings in the camel cased output should not contain punctuation marks.
  - This means that "Hello.World." gets split into 2 sentences "Hello" and "World" and NOT "Hello.World".
- Let the camelCasing of `input_s` be called `output_s`
- `output_s` is the the concatenation of all words (`w`) in `input_s` after (http://linux.die.net/man/1/w) capitalized (http://linux.die.net/man/1/w)
  - The punctuation from `input_s` is **not** added to `output_s`.
- Words are:

    - nonempty substrings delimited by the MAXIMAL amount of whitespace.
        - This means that `" hello world "` is split into `"hello"` and `"world"` and NOT `"hello "`, `" "`, `" world"` or any other combination of whitespaces
    - considered uppercased if all of its letters are uppercased.
    - considered lowercased if all of its letters are lowercased.

    w
- a word (https://linux.die.net/man/1/w)
    - it is the first word and it is lowercased
    - it is any word after the first word and its first letter is uppercased
- Punctuation marks, whitespace, and letters are defined by `ispunct()`, `isspace()`, and `isalpha()` respectively.
    - These are parts of the C standard, so you can `man ispunct` for more information.
    - If `input_s` has ANY non-{punctuation, letter, whitespace} characters, they go straight into `output_s` without any modifications. **ALL** ASCII characters are valid input. Your camelCaser does not need to handle all of Unicode.
- `camel_caser` returns an array of `output_s` for every `input_s` in the input string, terminated by a NULL pointer.

Hint: `ctype.h` has a lot of useful functions for this.

Your implementation goes in camelCaser.c, and you may not leak any memory.

Destroy

You must also implement `destroy(char **result)`, a function that takes in the output of your `camel_caser` and frees up any memory used by it. We will be calling this in our test cases and checking for memory leaks in your implementation, so remember to test this!

Camel Caser Result In Memory

camelCaser takes in a C string, which represents an arbitrary number of sentences, and returns a pointer to a NULL-terminated array of C strings where each sentence has been camelCased. It is up to you how the resulting structure is allocated, but it must be completely deallocated by your `destroy` function.

For those who like pictures, here is what the return value of camelCaser looks like in memory:

 Memory Map

In the above picture, you can see that we have a `char` (https://linux.die.net/man/3/char) pointer called array. In this scenario, the `char` (https://linux.di pointer points to the beginning of a NULL-terminated array of character pointers. Each of the character pointers in the `char` array points to the beginning of a NULL-terminated (https://linux.die.net/man/3/char) array that can be anywhere in memory.

These arrays are NULL-terminated because your user will need to know when these arrays end so they do not start reading garbage values. This means that `array[0]` will return a character pointer. Dereferencing that character pointer gets you an actual character. For demonstration purposes, here is how to grab the character “s” in “as”:

```
// Take array and move it over by 3 times the size of a char pointer.
char **ptr = array + 3;
// Deference ptr to get back a character pointer pointing to the beginning of "as".
char *as = *ptr;
// Take that pointer and move it over by 1 times the size of a char.
char *ptr2 = as + 1;
// Now dereference that to get an actual char.
char s = *ptr2;
```

Writing Unit Tests

Your goal is to show that the other interns' implementations of camelCaser - which, of course, you can't see directly - fail on some extreme test cases, and, in the meantime, demonstrate to the head honcho at Facenovel exactly how robust your own function is.

Facenovel promises to pass in C-strings. Likewise, you promise to return a dynamically allocated NULL-terminated array of strings that can be deallocated with your `destroy` function.

What kinds of edge cases might come up?

Run `make camelCaser` to test. You will have to fill in tests in `camelCaser_tests.c`.

Because Facenovel values their testing server time, you may not try more than 16 different inputs, and each input must be less than 256 characters (only characters). This does NOT mean your implementation can assume input of 256 characters or less.

Also, it is not in the spirit of unit testing to diff the output of your implementation with the one you are testing. Therefore, you may **NOT** call your own `camel_caser` function when implementing your test cases.

Other helpful resources: Test-Driven Development (http://wiki.c2.com/?TestDrivenDevelopment)

Reference Implementation

Your senior coworkers at Facenovel have taken a liking to you for your work ethic, and they decided to help you by providing you a *reference implementation*. You are given an interface `camelCaser_ref_tests.c` which allows you to access the black-box reference implementation. You are given two utility functions to help you understand what camelCase looks like.

The first function provided is `print_camelCaser(char *input)`. It takes a string input and prints out the transformed camelCased output onto `stdout` (https://linux.die.net/man/3/stdout). This function is meant to help you answer questions like, "What should be the result of inputting `<blah>` into `camel_caser()`?" Note that this function might behave weirdly with non-printable ASCII characters. The exceptions are `\a` and `\b`, which we have escaped in the output so that it displays on the terminal. Take **extra care** when using it with `\a`, `\b` and `\\a`, `\\b`. (For more details, here's an article on escape sequences (https://en.wikipedia.org/wiki/Escape_sequences_in_C).

As an example, if you type this in `camelCaser_ref_tests.c`

```
char *input = "\aHello.";
```

this is displayed on the terminal

```
Test case
Input: Hello.
Output:
{
    "\ahello",
    NULL
}
```

and if you do this instead:

```
char *input = "\\aHello.";
```

this is displayed on the terminal instead

```
Test case
Input: \aHello.
Output:
{
    "",
    "ahello",
    NULL
}
```

We parse the `\a` and `\b` so that it gets displayed on the terminal. You **should not** be doing this in your camelCaser implementation.

The second function that you can use is `check_output(char *input, char **output)`. This function takes the input string you provided and the expected output camelCased array of strings, and compares the expected output with the reference output. The function returns `1` if the expected output is exactly the same as the reference output, and `0` otherwise. This function is to be used as a sanity check, to confirm your understanding of what the camelCased output is like.

A few important things to be aware of when you use the reference implementation:

- **DO NOT use any functions** provided in `camelCaser_ref_tests.c` and `camelCaser_ref_utils.h` in any part of your `camel_caser()` implementation as well as your unit tests (that is, don't use it anywhere outside of `camelCaser_ref_tests.c`). Your code will definitely not compile during grading if you use any of those functions in any graded files.
- The reference only serves as a starting guideline and a sanity check, to ensure that you understand how camelCase works. The reference implementation does not represent the only possible good implementation. **Your implementation is restricted by the specifications provided above, and only the specifications above.** An implementation is good if and only if it meets the requirements in the specifications.
- The reference **does not replace actual testing** of your own implementation. You are responsible to rigorously test your own code to make sure it is robust and guards against all possible edge cases.

To use the reference, modify the `camelCaser_ref_tests.c` file, run `make camelCaser_ref` and you should have a `camelCaser_ref` executable.

Note: The reference implementation is only available in release form. There is no debug version of the reference.

Grading

Grading is split up into two parts.

Your Implementation

The first portion of the test cases test *your implementation* of `camel_caser`. We pass in some input, and check that your output matches the expectations laid out in this document. Essentially, your code is put up against **our** unit tests, which means you can write as-good (or even better) unit tests to ensure that your `camel_caser` passes ours.

Your Unit Tests

The second portion of the test cases test *your unit tests*. We have a handful of `camel_caser` implementations, some that work, and some that don't. To test your unit tests, we feed each of our `camel_caser` implementations through your `test_camelCaser` function (found in `camelCaser_tests.c`) and see if it correctly identifies its validity.

For each of the `camel_caser` implementations that you correctly identify - marking a good one as good, or a bad one as bad - you get a point. To prevent guessing, randomization, or marking them all the same, any incorrect identification **loses** you a point. However, after each autograde run, we *will* tell you how many good ones you correctly identified and how many bad ones you identified, so you know which unit tests may need improvement.

If your unit test segfaults or otherwise crashes, our test will interpret that as evaluating that implementation as a bad one.

You cannot assume *anything* about the input, other than the input string being NULL-terminated, just like all good strings in C.

Example

Let's say there are five good implementations and five bad implementations. If you correctly say all five good are good, then you'd get +5 points. If you correctly identify three of the bad ones as bad, then you would get +3 for those and -2 for incorrectly labeling the others. In this case, you'd get a 6/10.

Good luck!