Assignment: Pointers and Half-Open Ranges

Learning Outcomes

- Gain practical experience in processing character strings.
- Gain practical experience in using and manipulating pointers, pointer arithmetic, compact pointer expressions, and understanding the relationship between arrays and pointers.

Task

The objective of this exercise is to provide you with practical experience with the material covered in lectures this and the previous week: pointers, pointer arithmetic, character strings, and half-open ranges. Your program will scan through bytes performing various operations declared below:

```c
char const* find(char const *begin, char const *end, char val);
char const* find_any(char const *begin, char const *end,
                     char const *vals, int len);
int count(char const *begin, char const *end, char val);
int count_any(char const *begin, char const *end, char const *vals, int len);
int compare(char const *begin1, char const *begin2, int len);
void exchange(char *begin1, char *begin2, int len);
void copy(char *dst, char *src, int len);
```

Function `find` returns a pointer to the first occurrence of `val` in a half-open range of values. If no such element is found, the function returns `NULL`.

Function `find_any` returns a pointer to the first occurrence of any member of the array whose first element is pointed to by pointer `vals` and has `len` number of elements in a half-open range of values. If none of the members of array `vals` are found, the function returns `NULL`.

Function `count` returns the number of elements in a half-open range of values equivalent to `val`.

Function `count_any` returns the number of elements in a half-open range of values equivalent to any member of the array whose first element is pointed to by parameter `vals` and has `len` number of elements.

Parameters `begin` and `end` in the above four functions represent a half-open range of array elements. Parameter `begin` points to the first element in the range while parameter `end` is a *past-the-end* pointer pointing to the element after the last element in the range. Thus, `begin` and `end` define a *half-open range* `[begin, end)` that includes the first element [pointed to by `begin`] but excludes the last element [pointed to by `end`].

Function `compare` compares corresponding elements in the arrays whose first elements are pointed to by `begin1` and `begin2` [both of which have `len` number of values] to determine if they contain the same values in the same order. If the contents of the two arrays are exactly similar, the function returns `0`. The function returns a negative value when an element in the array whose first element is pointed to by `begin1` has a smaller value than the corresponding element in the array whose first element is pointed to by `begin2`. Otherwise, the function returns a positive number.

Function `exchange` swaps the values in the two arrays whose first elements are pointed to by `begin1` and `begin2` with both arrays containing `count` number of elements.

Function `copy` copies `len` number of values starting from the element pointed to by `src` into corresponding elements pointed to by `dst`. It is possible that the two ranges overlap, so you must handle this scenario carefully! This means that you'll need to think and solve the problem on paper before writing any code. Notice that both pointers are not `const` qualified, meaning that you're modifying the arrays in-place. You're not allowed to create any new arrays in the code.

Implementation Details

Caveats

There are a few caveats your implementation must satisfy:

> ***Your submission cannot use the subscript operator `[]`. Instead, only pointer offsets and pointer notation is allowed. This means that your source code should not contain neither the `[` nor `]` symbol [even in comments].***
>
> ***Your submission must not rely on C standard library functions. This means both `q.c` and `q.h` must not contain any `include` directives [even in comments].***
>
> ***Your code must not define any arrays. This is simply because you don't need to. If you think you need one, then you don't understand the assignment. Ask questions if you're unable to solve the problem without defining arrays.***
>
> ***Pay attention to the `const`ness of function parameters. If a pointer is `const` qualified, then you're only reading the data that is pointed to; you won't be writing any data. If the pointer is NOT `const` qualified, then it'll likely be pointing to random garbage and the intent is for you to write useful data to the memory.***

Strategy

1. Begin by authoring a header file `q.h` that declares the seven functions listed here.
2. Author *stub functions* for the functions declared in `q.h` in source file `q.c`. A *stub* is a skeleton of a function that is called and immediately returns. It is syntactically correct - it takes the correct parameters and returns the proper values. The stub function for function `find` would look like this:

```c
// return any value of type char const* to ensure that the definition is
// syntactically correct and will compile although with diagnostic
// warning messages because parameters begin, end, val are unused.
// The most appropriate pointer value to return is NULL.
char const* find(char const *begin, char const *end, char val) {
  return NULL;
}
```

Although a stub is a complete function, it does nothing other than to establish and verify the linkage between the caller and itself. But this is a very important part of coding, testing, and verifying a program. At this point, the source file should be compiled.

3. Download the *makefile* provided for this assignment. Run `make` only with the rule to create object file `q.o` from source file `q.c`:

   ```
   $ make q.o
   ```

   Since you've only defined stub functions, parameters in these stub functions are unused causing the `-Werror` option to terminate compilation [with *unused parameter* error messages]. Therefore, with stub functions, you will need to temporarily drop the `-Werror` option in *makefile* to successfully compile [but with warnings]. Make a mental note to add the `-Werror` option to run the unit tests and prior to submission.

4. Download the driver source file `qdriver.c` containing a comprehensive set of unit tests for each of the seven functions to be implemented. Now, you've all the necessary files required to build executable `q.out` using the *makefile*:

   ```
   $ make
   ```

   Chances are that you'll find some problems related to syntax such as missing semicolons or errors between function prototype declarations and the function definitions. To enable `make` to build `q.out`, you should correct these problems.

5. At this point, you've `qdriver.c` and you've defined stub functions in `q.c`. Begin by implementing function `find`. The seven functions are declared in increasing order of difficulty - my recommendation is to implement the functions in the order they're declared. If you're designing algorithms before defining these functions [as you should], you will realize that you can implement the algorithm for `find_any` using the algorithm for `find`. This is called *code reuse* and programmers must strive to do this as much as possible. This idea of reuse is codified in the software engineering community through the DRY principle.

   > ***In previous assignments, we did not have to include `q.h` in `q.c` - there wasn't any need to. In this assignment, to satisfy the DRY principle, you'll perceive the need to include `q.h` in `q.c` so that you can define functions such as `find_any` that rely on other functions defined in `q.c` such as `find`. Don't!!! Instead, think about how to order the definitions of functions in `q.c` so that function `find` is declared [and also defined] before the compiler encounters the definition of function `find_any`.***

6. Before implementing other functions, you must verify that function `find` behaves correctly. To test a function, you should know what input is given to the function and the expected output from the function. The driver in `qdriver.c` provides 7 tests that are implemented by functions labeled `test1` through `test7` with `test1` containing unit tests for `find` - the first listed function - with `test2` containing unit test for the second listed function `find_any`, and so on. Users can specify a value between 1 and 7 through the command-line to invoke the corresponding test function. The driver will invoke all 7 test functions if the command-line value is 0. In addition, users must use the command-line to specify the name of the file that will contain the driver's output. If you wish to execute the tests implemented by function `test1` and save the driver's output to text file `myoutput1.txt`, you would then invoke the executable like this:

```
$ ./q.out 1 myoutput1.txt
```

A corresponding output file [containing the correct program output for the unit tests] is provided. Output file `output1.txt` corresponds to the unit tests specified for user input 1, and so on. Output file `alloutput.txt` contains the correct output generated when the driver executes all 7 unit tests - this happens when the driver is given the value 0 through the command-line. Use the `diff` command to compare your output and correct program output:

```
$ diff -y --strip-trailing-cr --suppress-common-lines myoutput1.txt
output1.txt
```

The unit tests in `qdriver.c` are comprehensive and will be used by the online auto grader to evaluate your submission.

File-level and Function-level documentation

Every source and header file you submit *must* contain file-level documentation blocks whose purpose is to provide human readers [yourself and other programmers] useful information about the purpose of this source file at some later point of time

Every function that you declare in a header file [and define in a corresponding source file] must contain a function-level documentation block.

> ***Don't copy and paste documentation blocks from previous assignments. Annoyed graders will definitely subtract grades to the full extent specified in the rubrics below when they detect such copy-and-paste scenarios.***

Submission and automatic evaluation

1. In the course web page, click on the submission page to submit `q.h` and `q.c`.
2. Read the following rubrics to maximize your grade. Your submission will receive:
    1. F grade if your submission doesn't compile with the full suite of `gcc` options [shown above].
    2. F grade if your submission doesn't link to create an executable.
    3. $A+$ grade if the submission's output matches the correct output. Otherwise, a proportional grade is assigned based on how many incorrect results were generated by your submission.
    4. A deduction of one letter grade for each missing documentation block. Every submitted file must have one file-level documentation block. Every function that you declare in a header file must provide a function-level documentation block. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and the three documentation blocks are missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the three documentation blocks are missing, your grade will be later reduced from C to F.