Tutorial 9

Strings, arrays, pointers, structures and file I/O

Learning Outcomes

- Develop skills in C programming with strings, arrays and pointers
- Practice use of file I/O and structures

Overview

The main focus of this lab task is implementing a rudimentary spell-checker. The spell-checker will assess words for correct spelling based on a selected dictionary file, but also include a few other features.

There are five functions that you need to implement to complete the task. They may return a valid output result or one of the special values represented by constants defined in the header file. The header file `q.h` will have the following contents in it. *Please do not modify it* :

```
typedef int WordCount;
typedef unsigned char WordLength;
typedef int ErrorCode;

struct DictionaryInfo
{
    // Length of the shortest word
    WordLength shortest;
    // Length of the longest word
    WordLength longest;
    // Number of words in the dictionary
    WordCount count;
};
typedef struct DictionaryInfo DictionaryInfo;

enum ErrorCode
{
    FILE_OK = -1, // The file was opened successfully.
    FILE_ERR_OPEN = -2, // The file was not opened.
    WORD_OK = -3, // The word was found in the dictionary.
    WORD_BAD = -4 // The word was not found in the dictionary.
};
enum
{
    LONGEST_WORD = 50
};

char* str_to_upper(char* string);
WordCount words_starting_with(const char* dictionary, char letter);
ErrorCode spell_check(const char* dictionary, const char* word);
ErrorCode word_lengths(const char* dictionary, WordCount lengths[],
```

```
                        WordLength count);
ErrorCode info(const char* dictionary, DictionaryInfo* info);
```

The functions are:

1. `char* str_to_upper(char* string);`

   Given a string, converts all lowercase letters to uppercase, and returns a pointer to the first character of the string. The function works *in-place*, which means it modifies the array that was passed in as a function parameter, without allocating a new array; the return object is the same value that was passed into the function, similar to `strcpy` and `strcat`.

   Function call to `test1()` from `main()` in `qdriver.c` can be used to test this function code.

2. `WordCount words_starting_with(const char* dictionary, char letter);`

   Given the file name of a dictionary text file, counts the number of words that start with a given letter. The function returns:

   - `FILE_ERR_OPEN` if the file cannot be opened;
   - otherwise, the number of words that start with the letter.

   The function matches characters in a *case-insensitive* way; you need to account for this. A way to do this is to make both operands of the comparison to uppercase with the helper function `str_to_upper()`. This helper function converts each lowercase character in a string to its corresponding uppercase character.

   Remember to close the opened file when you are done with it; not closing files is a **serious resource leak** in your programs.

   All function calls to `test3()` from `main()` in `qdriver.c` can be used to test this function code.

3. `ErrorCode spell_check(const char* dictionary, const char* word);`

   Given the file name of a dictionary text file and a word, looks up the word in the dictionary. The function returns:

   - `FILE_ERR_OPEN` if the file cannot be opened.
   - `WORD_BAD` if the word was not found.
   - otherwise, `WORD_OK`.

   The function matches characters in a *case-insensitive* way; you need to account for this using the same approach as for `words_starting_with()` function.

   All function calls to `test4()` and call to `test5()` and `test6()` from `main()` in `qdriver.c` can be used to test this function code.

4. `ErrorCode word_lengths(const char* dictionary, WordCount lengths[], WordLength count);`

   Given the file name of a dictionary text file, counts the number of words of each length between 1 and `count` (inclusive) and stores this result in an array `lengths` at the position corresponding to the length. The function returns:

   - `FILE_ERR_OPEN` if the file cannot be opened;
   - otherwise, `FILE_OK`.

   Since you are given a text file that contains lines of text, you can use `fgets()` to read them in. A new line character sequence makes `fgets()` stop reading, but it is still considered a valid character by the function and included in the produced string. However, in this program the new line sequence is not supposed to be included in the word. After reading the word from the file, you need to remove the new

line sequence from the word. On Linux the end of file sequence is `\n`, on Windows `\r\n`, on old Macintosh `\r`; to make the code portable you can simply replace the first instance of `\n` or `\r` (whichever comes first) with `\0`.

Function call to `test7()` from `main()` in `qdriver.c` can be used to test this function code.

5. `ErrorCode info(const char* dictionary, DictionaryInfo* info);`

   Given the file name of a dictionary text file, return its description (the length of the shortest and the longest words, and the count of all words) using the `DictionaryInfo` structure. The function returns:

   - `FILE_ERR_OPEN` if the file cannot be opened;
   - otherwise, `FILE_OK`.

   Use `fgets()` to read the the words in; remove the new line sequence from words using the same approach as for `word_lengths()` function.

   All function calls to `test2()` from `main()` in `qdriver.c` can be used to test this function code.

All of the functions, except `str_to_upper()`, work on a dictionary file. If they fail to open the file, they must return `FILE_ERR_OPEN`. In each dictionary file, there is exactly one word per line. All words end with a newline character which is not a part of a word; after reading the word from the file, you need to remove the newline from the word.

Open the provided file `lexicon.txt` to see an example of a dictionary file.

Task

1. Download the source code files:

   Go to the course page in *Moodle* - DigiPen (Singapore) online learning management system, download a zipped archive and extract its files into a Microsoft Windows directory `c:\sandbox\` (adjust the path as suitable for your system). The included files are:

   - `qdriver.c`
   - `expected-output.txt`
   - Dictionaries:
     - `allwords.txt` - the largest dictionary (100,000+ words), one of the input files,
     - `small.txt` - a dictionary with 35 words, one of the input files,
     - `lexicon.txt` - a dictionary with 12 words, one of the input files.

2. Using the Microsoft Windows command prompt navigate to the `sandbox` folder. Then type `wsl` to open Linux bash in the same current directory.
3. From Moodle download `q.h` and `q.c`. Study and copy the type aliases, macros, structure definition, and function declarations in the first page of this document into `q.h`.

   Also inspect other provided files, both the source code and the provided dictionaries.

4. Complete the definitions of all five required functions inside `q.c`.
5. Compile the program.

   ```
   gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic-errors -std=c11 -o
   q.out q.c qdriver.c
   ```

6. Run the executable `q.out`:

```
./q.out > actual-output.txt
```

7. Use `diff` to compare the actual output with the expected output:

```
diff -y --strip-trailing-cr --suppress-common-lines expected-output.txt actual-output.txt
```

   Make sure that the output matches **exactly**.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit files `q.h` and `q.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.
   - 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.