Tutorial 9

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

Purpose of the exercise

This exercise will help you do the following:

1. Develop skills in C programming with strings, arrays and pointers.
2. Practice use of file I/O and structure.

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 macros defined in the header file *spellcheck.h*:

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

// The file was opened successfully.
#define FILE_OK         ((ErrorCode)(-2))
// The file was not opened.
#define FILE_ERR_OPEN   ((ErrorCode)(-1))
// The word was not found in the dictionary.
#define WORD_BAD        ((ErrorCode)(0))
// The word was found in the dictionary.
#define WORD_OK         ((ErrorCode)(1))
```

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, similarly to `strcpy` and `strcat`.

   Call a function `test1()` from `main()` in *qdriver.c* to test the 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 uppercase with the helper function `str_to_upper()`.

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 the 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 calls to `test4()` `test5()` and `test6()` from `main()` in *qdriver.c* can be used to test the 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`.

   All function calls to `test7()` from `main()` in *qdriver.c* can be used to test the code.

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

   Given the file name of a dictionary text file, returns 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()`.

   All function calls to `test2()` from `main()` in *qdriver.c* can be used to test the 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 a dictionary file there is exactly one word per line in each dictionary. 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*
- *spellcheck.c*
- *spellcheck.h*
- *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 Linux open *spellchecker.h* in Microsoft Visual Studio Code.

   Study declarations of functions in this file. Also, inspect other provided files, both the source code and the provided dictionaries.
4. Complete the definitions of all five required functions inside *spellcheck.c*; you can find their declarations in *spellcheck.h*.
5. Compile the program.

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

6. Run the executable *spellcheck*:

```
./spellcheck > actual-output.txt
```

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

```
diff expected-output.txt actual-output.txt --strip-trailing-cr
```

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

Submitting the deliverables

You have to upload a complete file *spellcheck.c* to *Moodle* - DigiPen (Singapore) online learning management system where the file will be automatically evaluated.