Hack 10.0

Computer Science I
File I/O

Department of Computer Science & Engineering
University of Nebraska–Lincoln

Introduction

Hack session activities are small weekly programming assignments intended to get you started on full programming assignments. You may complete the hack on your own, but you are *highly encouraged* to work with another student and form a hack pair. Groups larger than 2 are not allowed. However, you may discuss the problems *at a high level* with other students or groups. You may not share code directly.

If you choose to form a Hack Pair, you *must*:

1. Both join a hack pair on Canvas (go to People then Hack Pairs)
2. You must both work on the hack equally; it must be an equal effort by both partners. Do not undermine your partner's learning opportunity and do not undermine your own by allowing one parter to do all the work.
3. Turn in only one copy of the code under the individual whose last name comes first (with respect to Canvas).

You are graded based on style, documentation, design and correctness. For detail, see the general course rubric.

Category	Point Value
Style	2
Documentation	2
Design	5
Correctness	16
Total	**25**

Table 1: Rubric

Correctness:

- 5 points for each function
- 1 point for error handling
- 5 points for protein program

Exercises

To get more practice working with files, you will write several functions that involve operations on files. In particular, implement the following functions.

1. Write a function that, given a file path/name as a string opens the file and returns its entire contents as a single string. Any endline characters should *be preserved.*

   ```
   char *getFileContents(const char *filePath);
   ```

2. Write a function that, given a file path/name as a string opens the file and returns the contents of the file as an array of strings. Each element in the array should correspond to a line in the file. Any end line character should be *chomped out* and not included. The size of the resulting array of strings needs to be communicated to the calling function using the pass-by-reference `numLines` parameter (it is *not* input).

   ```
   char **getFileLines(const char *filePath, int *numLines);
   ```

Protein Translation

DNA is a molecule that encodes genetic information. A DNA sequence is a string of nucleotides represented as letters A, T, C, and G (representing the nucleobases adenine, thymine, cytosine, and guanine respectively). Protein sequencing in an organism consists of a two step process. First the DNA is translated into RNA by replacing each thymine nucleotide with uracil (U). Then, the RNA sequence is translated into a protein (a sequence of amino acids) according to the following rules.

The RNA sequence is processed 3 bases at a time called a *codon*. Each codon is translated into a single amino acid according to known encoding rules. There are 20 such amino acids, each represented by a single letter in

$$(A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y)$$

Because there are $4^3 = 64$ possible codons but only 20 amino acids, some codons translate to the same amino acid.

The rules for translating trigrams are complex, but we've simplified the process by providing a utility function, `rnaToProtein` which takes an RNA codon (as a string) and returns its protein (as a single `char`). If you provide it an invalid sequence, it will return `\0` the null character.

In addition, the trigrams UAA, UAG, and UGA are special markers that indicate a (premature) end to the protein sequencing (there may be additional nucleotides left in the RNA sequence, but they are ignored and the translation ends). The function we've provided will return a lower-case `x` character for any of these three trigrams.

As an example, suppose we start with the DNA sequence $AAATTCCGCGTACCC$; it would be encoded into RNA as $AAAUUCCGCGUACCC$; and into an amino acid sequence $KFRVP$.

You will write a program that takes two command line arguments. The first is an input file containing a DNA sequence and the second is the name of the output file in which you'll place the translated protein sequence. As an example, your program should be executable from the command line as:

```
~>./a.out dnaInputFile.txt protein.txt
```

The input file *may* contain irrelevant whitespace characters to avoid very long lines. You will need to *ignore* any whitespace characters when you process the data.

Place all your code in a file named `proteinTranslator.c`.

Instructions

- For the exercises, place all your function prototypes into a file named `file_utils.h` and and their definitions in a file named `file_utils.c`. In addition, you'll want to create a main test driver program that demonstrates at least 3 cases per function to verify their output. You need not hand it in, however.
- Code for the `rnaToProtein` function as well as a demonstration on how to use it has been provided in the following repo:

  https://github.com/cbourke/CSCE155-Hack10.0

  However, you only need to handin `proteinTranslator.c` (and anything else that you may find helpful). Do *not* make changes to the provided files since you won't be able to hand them in (the grading script provides them for you).
- **Hint**: Code reuse is a Very Good Thing. Your protein program can use your file utility functions, but also: feel free to include any additional functions you may have written before in the `file_utils.h` and `file_utils.c` files and use them in your protein translator program.
- You are encouraged to collaborate any number of students before, during, and after your scheduled hack session.
- You may (in fact are encouraged) to define any additional "helper" functions that

may help you.

- Include the name(s) of everyone who worked together on this activity in your source file's header.
- Turn in all of your files via webhandin, making sure that it runs and executes correctly in the webgrader. Each individual student will need to hand in their own copy and will receive their own individual grade.