CS2002 Week 7: C2 - Code Breaker

Jon Lewis (jon.lewis@st-andrews.ac.uk)

Due date: Wednesday 11th March, 21:00
25% of Practical Mark for the Module

Objective

To gain experience with programming in C and creating and manipulating suitable data abstractions.

Learning Outcomes

By the end of this practical you should:

- be capable of constructing simple data abstractions in C
- be capable of allocating and manipulating collection and record structures
- have extended your experience in writing modular C programs
- understand the need for thorough testing and providing evidence thereof

Overview

In this practical you will write a program that counts alphabetic and whitespace characters in text files on StudRes at

```
http://studres.cs.st-andrews.ac.uk/CS2002/Practicals/W07-C2/data/
```

You will also extend your character counter to decode pieces of encrypted cipher text. Sample cipher text files and key text files are also supplied in the same directory. Finally you will write a short report.

Character Counting

You are required to write a program to read in characters from a file the name of which is specified on the command line and output a table showing the count of the occurrences of each alphabetic or whitespace character in the file. Characters to be treated specially are:

- Alphabetic characters: For these, you should compute a combined count of both the upper and lower case instances. You may find a suitable alphabetic test for characters in `"ctype.h"` that you could use.
- Whitespace characters: take these to comprise space, tab, linefeed, carriage return, vertical tab, form feed, and you should compute a combined count of all whitespace characters. Once again, you may find a suitable whitespace character test in `"ctype.h"`.
- You must provide a Makefile with a default target (the first one) which builds your character counter executable as `CounterMain`. You should also provide a `clean` target in the Makefile to remove the executable and other `.o` files generated during the compilation process.
- Your `CounterMain` program should expect a filename to be passed as a command line argument to your main function (remember `argv[0]` is always the name of the executable and `argv[1]` is the first proper argument). It should count the characters in this file and print out the occurrences once it has reached the end of the file.

- All your source code for the project must be in a `src` directory within your assignment directory.

Below are some examples of executing your compiled program from within the `src` directory and pointing it at files in the `data` directory within your assignment folder. The examples show the expected output for some different command-line arguments, indicating the functionality that you should provide:

```
./CounterMain
Usage: ./CounterMain <some_file.txt> [key.txt]

./CounterMain ../data/non_existent_file.txt
Trying to open file: No such file or directory

./CounterMain ../data/Hello_World.txt
Total chars counted: 11
Char, Count
_, 1
d, 1
e, 1
h, 1
l, 3
o, 2
r, 1
w, 1

./CounterMain ../data/Hello_World_LF_CR.txt
Total chars counted: 13
Char, Count
_, 3
d, 1
e, 1
h, 1
l, 3
o, 2
r, 1
w, 1
```

The first two invocations above indicate what your program should print when no command-line arguments are supplied and when the specified file cannot be found. The other invocations specify files that do exist. The `Hello_World.txt` file simply contains the string "`Hello World`" and the `Hello_World_LF_CR.txt` file contains the same text, but with an additional linefeed and a carriage return character at the end. Note that the occurrence of these additional characters is treated as additional occurrence of whitespace characters, indicated as "_" in the program's output. After processing the file, your program is supposed to produce the output as shown above, containing a report on the total number of characters that were found in the file and a comma-separated list of the character counts (ordered by ASCII character value).

Method

You do not need to use dynamic memory allocation or pass by reference for this practical when designing your data types. Look at the Person example (L05/PersonStructByValue) on student resources as an example of providing abstract data types without the necessity of pass by reference or dynamic memory allocation. That said, dynamic memory allocation, and maintaining/passing pointers to structs (see L07-08/PersonStructDynamic) may improve modular design, style and efficiency.

You should design suitable data types to represent CharacterCount information relating to a single character and a data type to represent a Collection of these CharacterCount records, as well as the total number of characters counted. The Collection can hold a fixed size array of records. A neat way to index this array is to use the ASCII character values themselves as the indices. That is, you could hold the count record for character c in the element `store[c]` where `store` is an array of 256 records in your Collection type (or possibly an array of pointers to records if you choose to use dynamic memory allocation). But no magic numbers! Work out how to `#define` the size without actually doing any calculation yourself.

Provide suitable operations for your data types, such as creating a new collection data structure, updating the collection by changing the count for a given character that has been found in a file, and to perform any other necessary actions, such as printing out the collection. As for any module, provide an interface for this module in a corresponding header file. To aid your development, it is probably worth writing a short test program to verify that your modules work as intended.

Methods which change the content in your data structures will have a form similar to:

```
    Structure incrementCount(Structure s, char c);
```

where a new, updated struct is returned from the method. This is because the function `incrementCount` cannot change `s` unless it is passed in by pointer. Should you decide to use dynamic memory allocation and pass pointers to structs, then the method could have a form similar to:

```
     void incrementCount(Structure *s, char c);
```

as it no longer needs to return a copy of the updated structure.

In a separate file reader module, you should probably write functions that open a file specified by filename and return the next valid character from a specified stream. Try to consider erroneous input here and permit your program to exit gracefully if the specified file does not exist as shown in the output above.

In another module, you should write code that takes the filename passed into `main` and uses your file and Collection modules' functions to continually get a character from a file specified on the command line and to update the collection structure for a given character until the end of the file is reached. You should provide functions to populate, query and print out the collection. Finally, complete the program as specified, and test that it works. You may wish to create a few small text files of your own, e.g. a file with a few different characters, for initial testing before attempting to count the characters in the supplied text files. As always provide a Makefile to build your executable programs. Once you have a Makefile with default (first) target and clean target, you will be able to use stacscheck and run a number of functionality tests over your program (see section on Running stacscheck below).

General Programming Tips

Use appropriate header files for type declarations, function prototypes etc. and structure these properly. Use header guards (conditional compilation instructions) where necessary in `.h` files. Keep the main routine short, effectively a series of function calls. Devolve the work of the system into separate (small) functions, passing relevant parameters such as collection or record structures to functions. Look at the Person examples on student resources as examples of providing abstract data types.

Decoding

Once you have constructed your character counting program you can extend your program to break a simple code. It should take two files, called the cipher and the key. The cipher has the correct order of characters, but a substitution has been performed on them, whereas the key has the correct letters rearranged randomly. Extend your program to make use of the optional third command-line argument "`key.txt`" indicated in the usage message shown above. That is, your program will now treat the first command-line argument as the cipher file and the second argument as the key file. It should count the characters in both of them.

To simplify the problem the cipher only covers the 26 letters of the alphabet plus a whitespace character. All occurrences of various whitespace characters are conflated into a single count for a whitespace char. Numbers and any other characters are not involved and whether a letter is in upper or lower case is ignored. The cipher is a 1-1 mapping from plain to cipher text. As an example, given the true text string "`c a t at t`", a valid key is "`tat  .  Tac`" (two spaces before the "`.`" and two after), while a valid cipher is "`b h u hu u`". Note that in this particular case, the true space character is mapped to the space character in the cipher as well, although this need not be the case in general.

Given both a cipher and a key, your program should be able to calculate the original text. Your program should do this in phases:

1. Adapt your program to detect the presence of both cipher and key as command-line arguments, and then to subsequently calculate the character counts for both of them.
2. Potentially adapt your CharacterCount record struct from Part 1 to include a member that can record the original true-text replacement character for a particular cipher-text character, and potentially add functions to set the replacement.
3. Build a Collection which maps cipher-characters to original true-text characters, by looking for places where the character count for the cipher and key agree. You may assume that each count will be unique, except when 0.
4. Re-open and read the cipher file again character by character, but this time, print out to the screen the true-text replacement for each observed cipher-text character (as indicated by the Collection you built in the previous step), and thereby decode the file. Note your program should not have stored the cipher file in memory, you should reopen it and print it back out, but it should store in memory the Collection that maps from cipher characters back to the original text.

The key files in the `data` directory contain all the characters of the original plaintext message with the same frequency with which they occur in the plaintext. That is, the replacement cipher has not been applied to the characters in the key files, instead the order of the characters has been randomized to obscure the original plaintext message. Capitalisation, commas and full-stops have also been applied to the text to obscure the original message. The cipher files contain encrypted text.
Below is an example of executing your compiled program from within the `src` directory and pointing it at both cipher and key files in the `data` directory (assumed to be within your assignment folder). The example shows the expected output of decoding the `catatt_cipher.txt` file by using the `catatt_key.txt` file:

```
./CounterMain ../data/catatt_cipher.txt ../data/catatt_key.txt
Decoding
c a t at t
```

Running stacscheck

Some stacscheck tests have been provided to help you verify the operation of your program against much of the functional specification. You can run the automated checking system on your program by opening a terminal window connected to the Linux lab clients/servers and executing the following commands:

```
cd ~/CS2002/W07-C2
stacscheck /cs/studres/CS2002/Practicals/W07-C2/Tests
```

assuming `CS2002/W07-C2` is your assignment directory. This will run a number of character count tests over your program and also some decode tests.

Deliverables

Hand in via MMS, by the deadline of 9pm on Wednesday of Week 7, a zip file containing:

- Your assignment directory with all your source code, including any of your own tests.
- A PDF report describing your design and implementation, any difficulties you encountered, how you tested your implementation above and beyond using stacscheck. Take care to explain and justify your design and implementation decisions in clarity and detail.

Marking Guidance

The submission will be marked according to the general mark descriptors at:

`https://studres.cs.st-andrews.ac.uk/CS2002/Assessment/descriptors.pdf`

A very good attempt with decomposition of your code into a sensible set of modules and functions achieving almost all required functionality, together with a clear report showing a good level of understanding, can achieve a mark of 14 - 16. This means you should produce very good code with very good decomposition and testing and provide clear explanations and justifications of design and implementation decisions in your report. To achieve a mark of 17 or above, you will need to implement all required functionality. Quality and clarity of design, implementation, testing, and your report are key at the top end.

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof): `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`

Good Academic Practice

As usual, I would remind you to ensure you are following the relevant guidelines on good academic practice as outlined at

`https://www.st-andrews.ac.uk/students/rules/academicpractice/`