# CS2002 Week 7: C2 - Word Counter (25%)

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

Due date: Wednesday 9th March, 21:00
MMS is the definitive source for practical weighting and due date/time

## Objective

The aim of this practical is to develop a program to read in words taken from a file, and to produce a chart showing the frequency of those words.

The objective of this practical is to exercise the use of strings, basic data structures, reading from text files in C.

## Learning Outcomes

By the end of this practical you should:

- be familiar with basic string manipulation operations in C;
- be capable of reading textual data from a file;
- be capable of allocating and manipulating collection and record structures;
- have extended your experience in writing modular C programs.

## Getting started

You are expected to have read and understood the lecture notes on strings and arrays (including command-line arguments), structs, and file input/output, and to have completed the W03-Exercise work on basic C programming and separate compilation.

As with the week 3 exercise, I would probably recommend using VS Code as an editor to develop your code using a remote connection to the labs, such that your code will be stored and backed up on lab machines and you have a terminal window open to the lab machine. If you are unsure how to do this, there are some pointers in the W03-Exercise to tutorials on using VS Code at

`https://studres.cs.st-andrews.ac.uk/CS2002/Practicals/W03-Exercise/W03-Exercise.pdf`

In any case, you should create a suitable project directory. On the lab machines this could be in `Documents/CS2002/W07-C2` and your source code must be in a `src` directory within your project directory. You can do this easily from the command-line on the lab machines as shown below

```
mkdir -p ~/Documents/CS2002/W07-C2/src
```

You can download some text files to use when testing your solution to this practical from student resources at

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

However, more variants of these files are also available as part of stacscheck tests (see section on Running Stacscheck below).

## Overview

In this practical you will write a program to read in words from a file that is specified on the command line and output a chart showing the frequency with which each word occurs in the file, irrespective of upper or lower case. The order of the output is the order in which new (previously unseen) words are encountered in the file. Also, for the purposes of this assignment, a word is simply defined as a sequence of one or more letters (alphabetic characters).

- You may find a suitable alphabetic test for characters and a means to convert between upper and lower case in `"ctype.h"` that you could use.
- You may assume sensible maximum values both for the number of unique words contained within a file and for the word length.
- Your program should deal gracefully with exceptional situations such as not being able to open a file or successfully request additional memory (only if you make use dynamic memory allocation) or where your assumed maximum values are exceeded.
- You must provide a Makefile with a default target (the first one) which builds your word counter executable as `RunCounter`. You should also provide a `clean` target in the Makefile to remove the executable and other `.o` files generated during the compilation process.
- Your `RunCounter` 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 words 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.

Finally you will write a short report.

## Method

You do not need to use dynamic memory allocation or pass structs 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, maintaining/passing pointers to structs (see L07-08/PersonStructDynamic) may improve your design in terms of style and efficiency, as may use of dynamic memory allocation.

You should design and implement a suitable abstract data type to record frequency information for a single word. Define this as an independent module, with its own (`.h` and `.c` file) and with functions that can access / update the data in the ADT and can create a new record for a given word. Hint: you will need to make a copy of every new word that is input! The functions should be implemented in the `.c` source file. Provide an interface to this ADT in the corresponding `.h` header file. Include appropriate type definitions.

Design and implement a second data type to hold a collection of frequency records as another module. Provide operations to create a new collection and to update the collection for a given word (such as either by incrementing the frequency count for an existing word, or by creating and using a new frequency record for a previously unseen word) and to perform any other actions that are necessary.

Write a third module with the functionality to count words in a given file, which will read words from the specified file and enter them into the collection structure. As already mentioned above, a word is simply defined as a sequence of one or more letters (alphabetic characters) for the purposes of this assignment.

Finally, complete the program as specified and test that it works.

As you develop your implementation, you will also want to write some code (programs with their own main function that use your modules) to verify the functionality of your various modules and to ensure that each is working prior to moving on. When doing this you should really add targets for these small test programs to the Makefile so as to ease your own development work.

You must also add targets to the Makefile for the main executable `RunCounter` that links together all the appropriate `.o` files into a `RunCounter` executable. You will also need Makefile targets for each of those `.o` files, each of which compiles the corresponding `.c` source file to the `.o` file. The clean target in the Makefile should remove the `RunCounter` executable as well as all the `.o` files. Have a look at the Person examples on StudRes and the Makefiles there.

Testing your word counter as a whole is probably done most easily by using stacscheck (see below) and some tests are provided for you. To add your own tests you can copy the whole `W07-C2/Tests` directory from StudRes to your own directory and add sub-directories for your own tests, with suitable `input.txt` input files and `expected.out` files containing the expected result. Have a look at the ones on StudRes.

Instructions on "Compiling and Running" and "Running Stacscheck" and some "General Tips" are given below. Please do read these before starting, as they may help you avoid problems down the line.

## General Programming Tips

When implementing your data types, you may take inspiration from the Person examples on StudRes for CS2002 and even CS2001 code for an array-based collection should you wish to.

As mentioned above, you don't have to use dynamic memory allocation or pass by reference for this practical when designing your data types, although passing pointers to structs (see L07-08/PersonStructDynamic) may improve your design in terms of style and efficiency, as may use of dynamic memory allocation.

When reading from the file, you don't have to read everything in at once, or even line by line, reading alphabetic characters into a buffer, one by one from the stream and subsequently entering the word into the collection is a possibility.

## Compiling and Running

Assuming your assignment directory is as outlined at the start of this specification and you have a Makefile and implementation as outlined above, you should be able to build and run your word counter program on a file `test1.txt` by executing

```
cd ~/Documents/CS2002/W07-C2/src
make RunCounter
./RunCounter test1.txt
```

Running your program as above on `test1.txt` as given on StudRes and containing the text

```
   Mary had a little lamb,
 Its fleece was white as snow.
And everywhere that Mary went,
    The lamb was sure to go.

Anon.
```

should produce the output shown below in which frequency records are printed out in the order in which new (previously unseen) words were encountered in the file.

```
Word, Frequency
mary, 2
had, 1
a, 1
little, 1
lamb, 2
its, 1
fleece, 1
was, 2
white, 1
as, 1
snow, 1
and, 1
everywhere, 1
that, 1
went, 1
the, 1
sure, 1
to, 1
go, 1
anon, 1
```

## Running Stacscheck

Some stacscheck tests have been provided to help you verify the operation of your program against some 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 ~/Documents/CS2002/W07-C2
stacscheck /cs/studres/CS2002/Practicals/W07-C2/Tests
```

assuming `CS2002/W07-C2` is your assignment directory. This will attempt to build your `RunCounter` and run this against some sample inputs to verify correctness. This includes checking that your program prints out the correct error messages when no filename is supplied as command line argument or the specified file cannot be opened for reading.

## 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 testing.
- A PDF report describing your design and implementation, any difficulties you encountered, how you tested your implementation above and beyond using the provided 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 and referencing (including referencing of program code) as outlined at

`https://info.cs.st-andrews.ac.uk/student-handbook/academic/gap.html`