CS 211: Computer Architecture, Spring 2012
Programming Assignment 1: Wordstat

Instructor: Thu D. Nguyen

1 Introduction

This assignment is designed to get you some initial experience with programming in C, as well as compiling, linking, running, and debugging a C program in our environment.

Your task is to write a C program called `wordstat` that reads a text file, finds all the unique words in the file, prints them in lexicographical order along with the total number of times each word appears (case-insensitive) and a count of different case-sensitive versions of the word.

A *word* is defined as any sequence of letters (A-Z, a-z) and digits (0-9) that starts with a letter, followed by 0 or more letters and/or digits. This definition is equivalent to the following regular expression:

([A-Z]|[a-z])([A-Z]|[a-z]|[0-9])*

Words in the input file correspond to the longest sequences that match the above expression when reading the file from beginning to end. Each unique word is case-insensitive. That is, "book" and "Book" and "bOOk" are all occurrences of the same word. However, they represent three different case-sensitive versions of the word "book".

As an example, running `wordstat` on a text file with the following content:

Some Bogus ?Random1> (random1modnar) [34ranDom1] (Random1) boGus-$con@tent.

should produce:

Word	**Total No. Occurrences**	**No. Case-Sensitive Versions**
bogus	2	2
con	1	1
random1	3	2
random1modnar	1	1
some	1	1
tent	1	1

2 Implementation

Implement a program called `wordstat` with the following usage interface:

`wordstat <argument>`

where <argument> is either the name of the file that `wordstat` should process, or `-h`, which means that `wordstat` should print out a help menu to guide the user on how to use the program.

As discussed above, when invoked with a valid file name, `wordstat` should find and output all the unique words in the file in lexicographical order, along with the total number of times each word appears (case-insensitive) and a count of different case-sensitive versions of the word.

We leave the choice of data structures and algorithms up to you. However, your implementation *must be reasonably efficient*. For example, maintaining an array of records and doing linear search through this array would be unacceptable.

You are allowed to use functions from standard libraries (e.g., `strcmp()`) but you cannot use third-party libraries downloaded from the Internet (or from anywhere else). If you are unsure whether you can use something, ask us.

We will compile and test your program on the iLab machines so you should make sure that your program compiles and runs correctly on these machines. You must compile all C code using the gcc compiler with the `-ansi -pedantic -Wall` flags.

3 Submission

You have to e-submit the assignment using Sakai. Your submission should be a tar file named `pa1.tar`. To create this file, put everything that you are submitting into a directory (folder) named `pa1`. Then, `cd` into the directory containing `pa1` (that is, `pa1`'s parent directory) and run the following command:

```
tar cvf pa1.tar pa1
```

To check that you have correctly created the tar file, you should copy it (`pa1.tar`) into an empty directory and run the following command:

```
tar xvf pa1.tar
```

This should create a directory named `pa1` in the (previously) empty directory.

The `pa1` directory in your tar file must contain:

- `readme.pdf`: This file should briefly describe the main data structures being used in your program, a big O analysis of the run time and space requirement of your program, and any challenges you encounter in this assignment
- `Makefile`: there should be at least two rules in your Makefile:
  1. `wordstat`: build your `wordstat` executable.
  2. `clean`: prepare for rebuilding from scratch.
- source code: all source code files necessary for building `wordstat`. Your source code should contain at least 2 files: `wordstat.h` and `wordstat.c`.

We will provide a small script that you can use to check for the above required items. You do not have to run it, but it might help you to check that you are handing in everything that we are asking for.

4 Grading Guidelines

4.1 Functionality

This is a large class so that necessarily the most significant part of your grade will be based on programmatic checking of your program. That is, we will build a binary using the Makefile and source code that you submitted, and then test the binary for correct functionality against a set of inputs. Thus:

- You should make sure that we can build your program by just running `make`.
- You should test your code as thoroughly as you can. *In particular, your code should be adept at handling exceptional cases.* For example, `wordstat` should *not* crash if the argument file does not exist.

Be careful to follow all instructions. If something doesn't seem right, ask.

4.2 Design

Having said the above about functionality, design is a critical part of any programming exercise. In particular, we expect you to write reasonably efficient code based on reasonably performing algorithms and data structures. More importantly, you need to understand the performance (time & space) implications of the algorithms and data structures you chose to use. Thus, the explanation of your design and big O analyses in the `readme.pdf` will comprise a non-trivial part of your grade. *Give careful thoughts to your writing of this file, rather than writing whatever comes to your mind in the last few minutes before the assignment is due.*

4.3 Coding Style

Finally, it is important that you write "good" code. Unfortunately, we won't be able to look at your code as closely as we would like to give you good feedback. Nevertheless, *a part of your grade will depend on the quality of your code.* Here are some guidelines for what we consider to be good:

- Your code is modularized. That is, your code is split into pieces that make sense, where the pieces are neither too small nor too big.
- Your code is well documented with comments. This does not mean that you should comment every line of code. Common practice is to document each function (the parameters it takes as input, the results produced, any side-effects, and the function's functionality) and add comments in the code where it will help another programmer figure out what is going on.
- You use variable names that have some meaning (rather than cryptic names like `i`).

Further, you should observe the following protocols to make it easier for us to look at your code:

- Define prototypes for all functions.
- Place all prototype, `typedef`, and `struct` definitions in header (.h) files.
- Error and warning messages should be printed to `stderr` using fprintf.