CS 211: Computer Architecture, Fall 2010
Programming Assignment 1: Wordstat
Due: 11:59 PM, 10/5/2010

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, and keep track of all the different **palindrome** words used in the file.

A *word* is defined as any sequence of upper and lower-case alphabetical letters (A-Z, a-z) that appears in the input file preceded immediately either by the beginning of the file or a non-letter character and followed immediately by the end of the file or a non-letter character. Words should be case-insensitive. That is, "book" and "Book" and "bOOk" are the same word.

A *palindrome* is a word that reads the same forward as it does backward. For example, "noon", "NoOn", "I" are all palindrome.

2 Implementation

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

`wordstat <option> <file name>`

where:

`<file name>` is the name of the file that `wordstat` should process.

`<option>` is "-h", "-l", "-w", "-p", or "-pf".

wordstat -h: Help for how a user can run the program and quit.
wordstat -l: number of lines.
wordstat -w: number of words.
wordstat -p: list of **palindrome** words in lexicographical order.
wordstat -pf: list of **palindrome** words in lexicographical order along with the number of times each word appears.

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

Some ?Random> (random12) weird-$con@tent.
Madam, I'm adam.

should produce:

wordstat -h input.txt:
Usage: ./wordstat <option> <filename>
Options:

-l : number of lines
-w: number of words
-p: palindrome's statistics.

wordstat -l input.txt: 2
wordstat -w input.txt: 10
wordstat -p input.txt:

I
M
MADAM

wordstat -pf input.txt:

I	1
M	1
MADAM	1

We leave the choice of data structures and algorithms up to you. 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 Cereal 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 your source code, Makefile, and readme.pdf in a folder named pa1, `cd` into the directory containing this folder, then 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 extract all the files that we are asking for below directly into the empty directory.

Your tar file must contain:

- readme.pdf: This file should have five sections:

  - Instructions to run your program.
  - Summary of design
  - Running time analysis
  - Space requirement analysis
  - Challenges encountered

- Makefile: there should be at least two rules in your Makefile:

  `wordstat` build your `wordstat` executable.

  `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

We plan to have a grading scheme as follows:

Functionality	80%
Document	10%
Coding styles	10%

4.1 Functionality

This is a large class so that necessarily a 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 Coding Style

Having said the above about functionality, it is also important that you write "good" code. Thus, *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.