

**CSI 333 – Programming at the Hardware/Software Interface – Fall 1999**  
**Programming Project 2 AND Lab Exercise 4**

**WARNING:** Read and follow the instructions herein about Revision Logs before you start this project. Otherwise, you might get ZERO credit or face more serious consequences.

## 1 Objectives:

1. Practice designing software logically organized into modules.
2. Practice implementing the logical organization by (a) coding each separate module interface in a separate header file, (b) each separate module implementation in a separate `.cpp` file, and (c) using `#include`'s to guarantee that interface declarations seen by both the implementation and users of each module are consistent.
3. Develop a very small string scanning, lexicographic comparison and substring printing libraries specialized for the application requirement that memory space is to be conserved by avoiding the copying of char arrays.
4. Practice the implementation and use of a sorting algorithm.
5. Write a "Makefile" and use `make` to automate safe and efficient software rebuilds.
6. Observe the operation of software on "realistically" sized input files.

## 2 Advice

This project is designed to be done after you did the modularization and Makefile practice of Lab Exercise 3. Therefore, if you didn't have your lab session yet, you should work on that Lab Exercise before putting much effort into this project. That Lab Exercise includes starting work on two of the modules required for this project. Notice that a simple `strread` module implementation code was given out in a lecture.

Before you attempt to write even one line of code, you must read and analyze the problem statement ("specification") to understand what your program is supposed to do, and the design and implementation plans to understand how to go about creating the program. Read the whole assignment and submit any questions to the lecture, the newsgroup (`sunya.classes.csi333`), a TA or the professor.

Lab Exercise 4 is given near the end of this handout. Exercise 4 will give you a start toward the module of specialized string operations needed for this project and help you use the `ddd` graphical interface debugger more effectively.

## 3 Problem

A key word in context (KWIC) index, sometimes called permuted index, is an alphabetic listing of **lines** of text (which could be book, article or section titles, or even the actual lines or sentences

of a poem or novel) in which each line is listed several times, once for each (significant) **word appearance** in the line. The particular word appearance for which the alphabetization is done is called the **key word**. So, the lines are listed so the key words are in alphabetic order.

Each **listing** is shifted so the key word begins in a particular column: For us, column 40.

Therefore, the user can scan the key words to find all the lines in which a key word appears. The context of each appearance of a key word is immediately visible so the user can choose or rule out particular lines according to his or her interest.

## 4 Some Details

Valid input lines will be at most 79 characters plus the newline. Your program should handle an excessively long input line by printing the message

```
Input line too long...exiting.
```

and then exiting immediately. Code to do this has been provided in one of the lectures.

It should also handle blank lines, or lines that are all blanks by ignoring them.

For simplicity, all words will be treated alike. (Production KWIC indexes omit indexing on connective words such as “a” and “the”, and might exclude excessively general words such as “computer”) Likewise, our definition of a **word appearance** is simplified but specified in a way that the software will still give useful results when applied to punctuated text.

A **word appearance** in a **line** is a sequence of characters that begins at a particular **place** in the line and satisfies 3 conditions:

1. The **word appearance** begins at the beginning of the line or right after a space character.
2. The **word appearance** consists solely of alphabetic characters.
3. The **word appearance** ends right before a non-alphabetic character or at the end of the line.

Remark: This permits a punctuation mark, apostrophe, digit, etc. to terminate a word appearance as well as a space or the end of line.

These specifications enable the software to output interesting results when run on the public domain electronic texts and indices of them published by Project Gutenberg (<http://promo.net/pg>). You can download a few of these for testing; and we will use some of these for grading purposes. Samples will appear in the Project2 subdirectory of the ECL class account directory.

### 4.1 Lexicographic Comparison Algorithm

1. If one of the words is empty, then the empty word is less than or equal to (LEQ) the other word.
2. Compare the numeric values after conversion to lower case of the first letters of the two words. If these values are different, the lexicographic order of the words is the numeric order of these values.
3. If the converted values are equal, the lexicographic order is determined by the rest of each word, after the first characters.

You can implement this either iteratively or recursively.

## 5 Example

### Input:

```
PowerPC: Apple's core
good apple, ... rotten core apple
MIPS, Pentium and PowerPC
PC Apples and my apple's penthouse
A penthouse; 9Hey..IAmNotAWordSinceIBeganWith9 apple ALongLine GetTruncated
```

**Output:**(Does not include the lines of numbers which are printed here only to indicate the column numbers.)

```
          1          2          3          4          5          6          7
1234567890123456789012345678901234567890123456789012345678901234567890
                                A penthouse; 9Hey..IAmNotAWordSinceIBega
Hey..IAmNotAWordSinceIBeganWith9 apple ALongLine GetTruncated
                                MIPS, Pentium and PowerPC
                                PC Apples and my apple's penthouse
                                PowerPC: Apple's core
                                good apple, ... rotten core apple
                                good apple, ... rotten core apple
                                PC Apples and my apple's penthouse
use; 9Hey..IAmNotAWordSinceIBeganWith9 apple ALongLine GetTruncated
                                PC Apples and my apple's penthouse
                                PowerPC: Apple's core
                                good apple, ... rotten core apple
tAWordSinceIBeganWith9 apple ALongLine GetTruncated
                                good apple, ... rotten core apple
                                MIPS, Pentium and PowerPC
                                PC Apples and my apple's penthouse
                                my penthouse; 9Hey..IAmNotAWordSinceIBeg
                                PC Apples and my apple's penthouse
                                PC Apples and my apple's penthouse
                                A penthouse; 9Hey..IAmNotAWordSinceIBeganW
                                MIPS, Pentium and PowerPC
                                PowerPC: Apple's core
                                MIPS, Pentium and PowerPC
                                good apple, ... rotten core apple
```

## 6 Non-functional Requirement

A non-functional requirement for software is a requirement that does not involve the input-output or “logical” behavior, but may apply to the software’s internal structure, implementation language, and, for us, **Performance and Memory Use**.

The non-functional requirement for this project is that memory use must be minimized so `kwic333` can operate on somewhat large input files, such as electronic texts from the Gutenberg

project. To achieve this, once a non-blank line is read and stored in memory in a free-store allocated a character array, NO COPIES of the line or words in it are to be stored in memory.

To achieve that, all substrings will be described by a pointer to the originally read line plus a PAIR of integers  $(i, j)$  where  $i$  represents the index into the array at which the substring starts, and  $j$  is the index immediately after which it ends. (This enables the empty string to be represented by an index pair with  $i = j$ ; it makes programming easier.)

Sorting algorithms and data structures for **word appearances** and sets of them to which sorting algorithms can be applied will be covered in lectures.

## 7 Lab Exercise 4

Lab exercise 4 will be to design and implement a simplified module that provides a data type representing each of the lines after it is first read in, and provides just a few of the operations on substrings of a line that are needed for `kwic333`.

One of these operations is to print a substring. Here's how to do it simply: Within a loop, print the characters one at a time with the standard C++ statement:

```
cout << X;
```

where `X` is a `char` type variable or expression. This is the recommended technique for `kwic333` to print every output line.

Another operation is to find the length of the string. Another is to locate the next non-blank character after a blank if any at or after a given index.

Lab exercise 4 will also instruct you to use the ddd debugger to view the operation of code that processes the characters of the string type for this project one at a time. The ddd debugger will display the **CONTENTS** and **ADDRESS** and **INDEX** for each array position as your code processes it.

For the inlab checkoff, you will show the TA your code within the ddd window and demonstrate how the debugger displays what's specified above. The code must show the interface for your string processing module defined in its header file, and both your string function's implementation file and the tester's implementation file `#include`'ing this header file.

Additional tips for using ddd will be provided in the Lab and on the Web when we finish writing them.

## 8 Revision Log and Submission Checklist

You must maintain a "revision log". The revision log is to be maintained in a file named `revisions.txt` which must be submitted with all your source files (**every** `.cpp` and **every** `.h` plus the `Makefile`) needed for us to build the software for testing

At the beginning of the `revisions.txt`, after your name, student ID number and TA name, write a comment for each time you (1) began writing a new version, (2) fixed a bug, and (3) concluded by testing that the current version is correct; the comment must specify the version number, date and approximate time.

**A submission without a revision log will receive ZERO credit, and logs found to be faked will be treated as evidence of CHEATING.**

A word to the wise: Run `make` and do a final test immediately before submitting your work

using turnin. **If you edit anything, DO NOT TURN IT IN** until after you remake the software and test it again. The remake will verify your edits (or attempts to restore old code, etc) did not introduce syntax errors.

**IF WE CANNOT BUILD YOUR SOFTWARE from sources only (no objects or executables) BECAUSE OF SYNTAX OR UNDEFINED SYMBOL ERRORS, YOU WILL GET NO CREDIT FOR PROGRAM FUNCTIONS**

## 9 Test Cases, Grading etc.

1. By Tuesday, Nov. 2 a directory of sample inputs and correct outputs for indicated versions will be published in the ECL class account at pathname `~csi333/Project2/Samples` This material will also be put on the class web site.
2. The due time is Monday, November 15, 9:00PM.
3. Late submissions will be accepted until Friday, November 19, 9:00PM 9:00PM (4.000 days). However, a lateness penalty factor equal to  $(10.0 - ND)/10.0$  where  $ND$  is the amount of time late, measured in days, will be multiplied into your score, computed using floating point arithmetic. This means the amount deducted for latness will begin at 0% and rise (almost) continuously to 40% during the late turnin period. Early submissions will be accepted but will not earn any bonus.
4. 70% of the score will come from evaluation of outputs on test cases. To get any of these points, your software must build to an executable named `kwic333` when we run `make kwic333` in a directory with all the `.cpp` and `.h` files that you had submitted. All object (`.o`) files and executable files will be deleted before this is done; and points will be deducted if any such files were submitted. Give it a final test before you submit it; do NOT open it with an editor AT ALL after the final test.
5. 30% of the score will come from a physical organization that reflects a logical design in which separate problems are solved by separate modules, and internal documentation:
  - Quality (clarity, accuracy, completeness, etc.) of revision history.
  - Consistant indentation.
  - Procedures/functions: What each does in terms of parameter register contents, return value, and action on any other data it uses.
  - Header files containing interface definitions and documentation only, and other CSI333 software engineering standards in the lecture notes.
  - The dependencies are accurately coded in the `Makefile`.
  - The `Makefile` must be “simple” in the sense of Lab Exercise 3. DO NOT SUBMIT a “template” from a previous or other course! If you are a real Make expert, creating a simple makefile for this project will be easy. If not, follow the directions of Lab Exercise 3.