CSE4102 Common Problem Background

The intent of this document is to describe a problem that can then be implemented in multiple programming languages throughout the semester. There are three aspects to the problem: Word Count Functionality (WCF); Word Frequency Functionality (WFF); and, Permuted Index Functionality (PIF).

WCF Aspect

The first aspect involves word count Functionality (WCF) that reads from the standard input via a main program that employs argc (number of inputs on the command line) and argv (the input itself from the command line). Specifically, argv[i] contains the input from the command line that is delineated by blanks. For the word count program main, argv would contain the list of file names on which to provide the word count operation. Thus, if the execution of the program was "a.out file1.txt file2.txt file3.txt" then argc would equal 3 and argv will be defined as: argv[0]="file1.txt"; argv[1]="file2.txt,"; and, argv[2]="file3.txt". The program below implements the majority of the functionality for a program that counts lines, words, and characters, terminating with EOF (control-D in unix). Note that the newline character may not be recognized in MS Windows, and that there is no guarantee that the program below is correct and compilable.

```
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
  int lines[10], words[10], chars[10];
  int i, tot_lines=0, tot_words=0, tot_chars=0;
  /* Need error checks here */
  for (i=0; i<(argc-1); i++)
    {
     lines[i] = count_lines(argv[i+1]);
     words[i] = count_words(argv[i+1]);
     chars[i] = count_chars(argv[i+1]);
    }

  for (i=0; i<(argc-1); i++)
    {
     tot_lines += lines[i];
     tot_words += words[i];
     tot_chars += chars[i];
    }

  printf(" **********The Results ****************\n");
  /* Both Individual and Total results */
   for (i=0; i<(argc-1); i++)
    {
      printf("%10d%10d%10d%20s\n",
           lines[i], words[i], chars[i], argv[i+1]);
    }
```

```
   printf("Totals for the Input are:\n");
       printf("Lines: %10d\nWords: %10d\nChars: %10d\n",
           tot_lines, tot_words, tot_chars);
}

int count_lines(f_name)
    char  f_name[];
{
  FILE *fp;
  char c;
  int  num_lines=0;
  fp = fopen(f_name, "r");

  while ((c=getc(fp)) != EOF)
    {
      while (c != '\n')
        c = getc(fp);
      num_lines++;
    }
  fclose(fp);
  return(num_lines);
}
int count_chars(f_name)
    char  f_name[];
{
/* You need to supply the appropriate code */
}

int count_words(f_name)
    char  f_name[];
{
/* You need to supply the appropriate code */
}

\subsection*{WFF Aspect}
```

Using WCF, the second aspect of the problem, **Word Frequency Functionality, WFF**, accepts a text file name as a command-line argument via a main program. WFP is invoked from the command line as: `WFP file1.txt` and has the following functions in its processing:

1. CreateBaseList: Read the file (first pass) and create a linked list of words (in their order of occurrence), with the frequency of each word set to 0.
2. CaclWordFreq: Read the file (second pass) and for each word identified, search the linked list, and when found, increment the word frequency by 1.
3. PrintList: Print out each word and its respective word frequency count in the order of appearance.
4. AlphaSortandPrint: Sort the linked list alphabetically (assuming that each word starts with a letter) and then invoke the PrintList function.
5. FreqSortandPrint: Sort the linked list in decreasing order of word frequency (ignoring the alphabetic order for words with the same frequency) and then invoke the PrintList function.

One way to support such a program would be to utilize linked structure using C like pseudo code as follows:

```
typedef struct
{
  string                  word;
  integer                 word_freq;
  struct  word_list       *next_word;
  struct  word_list       *prev_word;
} word_list;

struct    word_list       *HEAD;
FILE                      *F_PTR;
```

In the structure, word is for each word identified, word_freq is for the frequency (number of times) the word occurs in the input file, next_word points to the next list element in the list, and prev_word points to the previous list element. For the first word, prev_word = NULL; for the last word, next_word = NULL. Let's also assume that there is a global pointer HEAD that points to the first list element. Note that the asterisk means pointer. Given this structure, the function CreateBaseList would be pseudo-coded as below.

```
CreateBaseList()
{
    struct  word_list     *list_ptr;
    struct  word_list     *new_element;
    string                temp_word;
    boolean               found;

    temp_word = GetWord(F_PTR);
    if temp_word == NULL  // Empty Input File
      then
        return NULL;
      else
        { // create the first node of the list
          new_element = new word_list;
          new_element->word = temp_word; new_element->word_freq = 0;
          new_element->next_word = NULL; new_element->prev_word = NULL;
          HEAD = new_element;

          while ( (temp_word = GetWord(F_PTR)) != NULL) do
            { // First, find if the word is there or the end of list
              list_ptr = HEAD; found = FALSE;
              while ( ((list_ptr->next_word != NULL) and (NOT found)) do
                {
                  if list_ptr->word == temp_word)
                    then // if word is found - then exit the loop
                      found = FALSE;
                    else // otherwise, increment to next word
                      list_ptr = list_ptr->next_word;
                }
              // if the loop exited and found still FALSE then add
              // the new word, since it was not found on the list so far
              if NOT found then
                 {
                   new_element = new word_list; // create the next node of the list
                   new_element->word = temp_word;
                   new_element->word_freq = 0;
                   new_element->next_word = NULL;
                   new_element->prev_word = list_ptr; // points to the previous
```

```
                    list_ptr->next_word = new_element; // points to new word
                  }
              }
            return();
          }
}
```

Note that this uses a function called GetWord that gets the next word (String) and returns NULL if EOF is detected. Note that this is pseudo code - it takes liberties with actual syntax while fully illustrating the flow of control and execution required to add a new word to the end of a list if it is not already present in the list. At the end of this function, HEAD will point to a list of words that is in the input file (and may be NULL).

PIF Aspect - Defined in C and C++

The third aspect, permuted index functionality (PIF), leverages both WCF and WFF and is described in both C and C++. For PIF, the aspect must be able to handle an unknown number of files (one or more), which have an unknown amount of lines (zero or more), where each line has a maximum number of characters (256). The requirements of the PIF aspect are defined in three steps:

Step 1 : Process argv by identifying all of the files and then reading in all of the files and storing the content of the files in dynamic memory structures. This will require you to read in every file, creating a list of lines for each file, and tracking a list of files as well. This will be accomplished using the line_list and input_file structures that are defined in p1pBhdr.h below.

Step 2 : Write a word count program that counts the words, characters and lines and outputs the results in tabular form on a file-by-file basis. Again, you will use the line_list and input_file structures that are defined in p1pBhdr.h, traversing them to perform the line, word, and character counts. Note that if you think about what you have accomplished in Step 1, you may find that Step 2 is relatively easy.

Step 3 : Traverse the structures created in Step 1 and create a permuted index. A permuted index tracks, for a given word, cross-references to the (file, line) pairs that represent the occurrence of the word. There may be multiple occurrences of the word in a file, e.g., the word "hello" can be in (FileA, 12), (FileA, 17), etc. However, if a word occurs multiple times within the same line, there is only one entry for that word in the permuted index. The permuted index is supported by two structures - references a list which tracks the (file name, line number) for each word, and permuted_index which tracks the list of identified words (in alphabetical order) and links to references. Thus, as you identify new words and insert them into the index, you must perform list operations that insert at the front of the list, the middle of the list, or the end of the list. You will need to use the references and permuted_index structures define below, as well as the line_list and input_file structures.

Note that in support of Steps 2 and 3, what constitutes a "word" for your permuted index must be more clearly defined. In WFC and WFF, you could simply count words as delineated by blanks or newlines. For Step 2, and particularly for Step 3, you must more clearly identify "words" as sequences of characters and digits; you will need to parse out punctuation and control characters from your counting in Step 2, and for Step 3, this is even more critical, to identify meaningful words to form the permuted index. Note that sample code for PIF is in the zip file located on the web page. This includes the file ZZZBcode.c that has sample code for dynamically creating line_list and input_file instances using malloc, and linking them with one another.

```
/* The p1pBhdr.h file */

#define MAX_CHARS 256

/* The line_list structure contains, for each line of a file,
   the line itself (256 characters - max in Unix),
   a integer line_no, and pointers to the next line
   (next_line) and previous line (prev_line). */

typedef struct
{
  char                line[MAX_CHARS+1]; /* Allow for end of string */
  int                 line_no;
  struct  line_list   *next_line;
  struct  line_list   *prev_line;
} line_list;

/* The input_file structure contains, for each file, the name of the
   file as read from argv (f_name), a pointer to the list
   of lines (f_lines), and a pointer to the next file (next_file). */

typedef struct
{
  char                f_name[MAX_CHARS+1]; /* Allow for end of string */
  struct  line_list  *f_lines;
  struct  input_file *next_file;
} input_file;

/* The references structure tracks, for each word in the
   permuted index, the list of (file,line) pairs.  It  does
   so by having a link to the input_file entry (file_ptr)
   and the corresponding line of that file (line_ptr).
   The structure is self-referential with a next_ref pointer
   to allow all of the references to the word in all
   (file, line) pairs to be tracked.    */

typedef struct
{
  struct  input_file *file_ptr;
  struct  line_list  *line_ptr;
  struct  references *next_ref;
} references;

/* The permuted_index structure tracks the list of all words
   that occur in all files.  Each permuted index entry contains
   the word found (in some line), a list of indexes to that
   word (the index_list pointer to the references structure),
   and a pointer to the next word in the permuted index
   (next_word pointer).  The words in this list must be
   kept in alphabetical order.  This list is created by
   scanning every line of every file, as tracked in the
   input_file structure.  */

typedef struct
{
  char                    word[MAX_CHARS+1];
```

```
  struct  references      *index_list;
  struct  permuted_index  *next_word;
} permuted_index;
```

The following transitions from a C to a C++ implementation. The next part provides a C++ implementation. The transition is one form where one is using a linked list and dynamic memory allocation in C to one where you can take advantage of targeted C++ application programmer interfaces (APIs) that encapsulate the gory details. Specifically, C++ APIs: `string` for string processing, `set` can be utilized to manage the permuted index, and `list` to manage any other linked data structures that are needed. Below, the .h files for the C++ classes, and the files Word.cpp (that goes with Word.h) and p1pCcode.cpp (for the main program) are shown; the complete code is in a zip file on the web page.

```
//************ THE FILE Word.h ********************

class CWord
{
public:
CWord(string str_word){ word = str_word; };
~CWord(void);
void insert_ref(CInputFile inpf, CFileLine fl);
bool operator< (CWord w2) const;
friend ostream& operator<< (ostream& os, CWord w);

private:
string word;
list<CReference> ref_list;
};


//************ THE FILE Word.cpp ********************

#include "Word.h"

CWord::~CWord(void)
{
}

void CWord::insert_ref(CInputFile inpf, CFileLine fl){
CReference tmp_ref(inpf, fl);

ref_list.push_back(tmp_ref);
}

bool CWord::operator < (CWord w2) const { return (word < w2.word); }

ostream& operator<< (ostream& os, CWord w){
list<CReference>::iterator r_iter;

os << w.word;
os << "\t\t";
for(r_iter = w.ref_list.begin(); r_iter != w.ref_list.end(); r_iter++){
  os << ((CReference) *r_iter);
}
```

```
return os;
};

//************ THE FILE FileLine.h ****************

#pragma once
#include <iostream>
#include <string>
#include <string.h>
using namespace std;

class CFileLine
{
public:
CFileLine(){};
CFileLine(string str_line, int lineno);
~CFileLine(void);
string get_line(){ return line; };
int get_lno(){ return line_no; };
int get_lwords(){ return l_num_words; };
int get_lchars(){ return l_num_chars; };
void count_words();
void count_chars();

private:
string line;
int line_no;
int l_num_words;
int l_num_chars;
};

//************ THE FILE InputFile.h ***************

#pragma once
#include <iostream>
#include <fstream>
#include <string>
#include <list>
#include "FileLine.h"

class CInputFile
public:
CInputFile(void){};
~CInputFile(void);
CInputFile(string fname);
string get_fname(){ return f_name; };
int get_numlines(){ return num_lines; };
int get_numwords(){ return num_words; };
int get_numchars(){ return num_chars; };
void parse_lines();
CFileLine get_line(int line_no);
friend ostream& operator<<(ostream& os, CInputFile inpf);

private:
list<CFileLine> l_lines;
```

```
string f_name;
int num_lines;
int num_words;
int num_chars;
};

//************ THE FILE Reference.h ***************

#pragma once
#include <iostream>
#include <string>
#include "InputFile.h"
#include "FileLine.h"

class CReference
{
public:
CReference(CInputFile inpf, CFileLine fline);
~CReference(void){};
string get_fname(){ return fp.get_fname(); };
int get_lineno(){ return fl.get_lno(); };
friend ostream& operator<<(ostream& os, CReference r1);

private:
CInputFile fp;
CFileLine fl;
};

//************ THE FILE PermIdex.h ****************

#pragma once
#include <iostream>
#include <string>
#include <set>
#include "Word.h"
#include "InputFile.h"

class CPermIndex
{
public:
CPermIndex(){};
CPermIndex(list<CInputFile> f_list);
~CPermIndex(void){};
void create_index(list<CInputFile> f_list);
friend ostream& operator<<(ostream& os, CPermIndex p1);

private:
set < CWord, less<CWord> > w_set;

};


//************ THE FILE p1pCcode.cpp ****************

#include <iostream>
#include <string>
```

```
#include <list>
#include "PermIndex.h"
#include "InputFile.h"
using namespace std;

int main(int argc, char* argv[]){
list<CInputFile>::iterator inpf_iter;
list<CInputFile> inpf_list;
CInputFile *tmp_f;
CPermIndex *perm_idx;
int i, tot_lines = 0, tot_words = 0, tot_chars = 0;

for(i = 1; i < argc; i++){
tmp_f = new CInputFile(argv[i]);
inpf_list.push_back(*tmp_f);

tot_lines+= tmp_f->get_numlines();
tot_words+= tmp_f->get_numwords();
tot_chars+= tmp_f->get_numchars();
}

cout << " **********The Results ****************\n";
for(inpf_iter = inpf_list.begin(); inpf_iter != inpf_list.end(); inpf_iter++)
cout<< ((CInputFile) *inpf_iter);

cout << "Totals for the Input are:\n";
cout << "Lines: " << tot_lines << endl;
cout << "Words: " << tot_words << endl;
cout << "Chars: " << tot_chars << endl;

perm_idx = new CPermIndex(inpf_list);

cout << *perm_idx;

}
```

For the entire problem, work with the same set of assumptions, namely:

1. Lines are terminated by a newline character. The last line of any file can be terminated by a newline/EOF combination, or simply an EOF. If there is only one line in a file (a good test case), that means that the file could be the single line with an EOF and no newline.

2. Words for the word count (and for the permuted index) must take into consideration the following: Each word must be at least one character and start with a letter. If a word has 2 or more characters, then the second and successive characters can be letters, digits, the underscore, or the hyphen. This means that you must, for the purposes of computing the permuted index, recognize and discard other characters. Also, note that you must eliminate white space (multiple spaces or tabs) between words.

3. For the purposes of the permuted index, please ignore case. That means that the following are all equivalent: red, RED, REd, ReD, etc.

For the different programming projects this semester, you will implement WCF, WFF, and/or PIF in different programming languages . You will need to take advantage of the different language capabilities (arrays, structures, classes, pointers, functions/procedures, modules, packages, etc.) as you repeat this project on different programming languages.