Assignment 5 : A Form Letter Processor

1. Description

In this assignment, you will write functions for a form letter producing program. In doing so, you will work with files and data storage and retrieval. The main loop of the program and some of the command line processing is provided; you will write the parts that do the work and expand the command line processing section of main().

The program is called **fl** for formletter. It is given two files: a format and a data file. These files look like:

```
sample format file:

     Dear %title% %ln%,

     You and everyone at %address% will be delighted to hear
     that you may already have won a major prize in the
     CSCI sweepstakes.  Yes, %fn%, you may be the
     person in %city% to be the lucky winner of a new jet
     aircraft, ocean liner, or luxury vacation.  Read on..
```

sample data file

```
title=Mr.;fn=John;ln=Doe;address=123 Main Street;city=Anytown
title=Ms.;fn=Jane;ln=Buck;city=Boston;address=47 Cherry;phone #=495-2000
```

The program reads the data file record by record. Each record is a single line containing one or more items of data. Each item consists of a fieldname, an equals sign, and the value for that field. The fields appear in no particular, nor consistent order. The fields are separated by the semicolon character. The end of the record is a single newline. (In the final version of this program, record and field separators can be specified by command line options.)

The program reads in a record and then plugs the data in that record into the form letter template. The template, as the example above shows contains regular text and special places marked by %fieldname% . These special places are replaced with the values given for that person. Processing continues until all input records are read.

2. The Code Framework

The framework for the program is provided for you. This consists of main(), a function called process(), and some data structure definitions. Main looks like:

```
#include  <stdio.h>
#include  "fl.h"

/*
 *    formletter program version 1.0
 *
 *    usage: fl format < datafile
 *
 *    data comes from stdin, output goes to stdout
 */

static char     *myname ;       /* used by fatal() */

int main(int ac, char *av[])
{
     FILE *fpfmt;

     myname = *av;

     /*
      *    check that there is at least one arg: the format file
      */

     if ( ac == 1 )
          fatal("usage: fl format [datafile..]","");

     /*
      *    then try to open it
      */

     if (  (fpfmt = fopen( av[1] , "r")) == NULL )
          fatal("Cannot open format file:", *av);

     /*
      *    in full version, code to handle names of data files
      *    on the command line will appear here
      */

     /* ... process data from stdin ... */

     process(fpfmt, stdin);
     return 0;
}

fatal(char *s1, char *s2)
/*
 *    fatal error handler
 *    purpose: print error message to stderr then exit
 *    input:   two strings that are printed
 *    never returns
 */
{
     fprintf(stderr, "%s: %s%s\n",  myname, s1, s2 );
     exit(1);
}
```

The actual work of the program is done by the function ‘process()’. This function looks like:

```
#include   "fl.h"
#include   "ws13.h"

/**
 *    process(fmt, data)
 *
 *    Purpose: read from datafile, format and output selected records
 *    Input:   fmt         - input stream from format file
 *          data     - stream from datafile
 *    Output:  copied fmt to stdout with insertions
 *    Errors:  not reported, functions call fatal() and die
 **/
process(FILE *fmt, FILE *data)
{
     symtab_t *tab;

     if ( (tab = new_table()) == NULL )
          fatal("Cannot create storage object","");

     while ( get_record(tab,data) != NO )/* while more data   */
     {
          mailmerge( tab, fmt );         /* merge with format */
          clear_table(tab);         /* discard data      */
     }
     free_table(tab);               /* no memory leaks!  */
}
```

3. The Data Structure Framework

Each record consists of one or more pairs of strings of the form: `city=Boston` . That is, a fieldname and a fieldvalue. Here the fieldname is ‘city’ and the fieldvalue is ‘Boston’. We already have software to store pairs of strings. The `wordstore.c` system provides a flexible system for recording and retrieving pairs of strings.

For this assignment, you will use a version called wordstore13.c . This version of the wordstore program includes a constructor to create a symbol table, and all the functions take as an argument a reference to a symbol table. You will call functions in that file to store and retrieve information.

You must use this storage system and this file. Once you get this level working, you will have to modify the main() program to allow the user to specify one or more data file names on the command line.

4. Part I: Adding Your Functions

The functions main() and process() shown above call various functions. You should write these functions to operate as described below:

`int get_record(symtab_t *tp, FILE *fp)`

get_record() reads the next data record from the stream at *fp. A record is a line containing one or more fields of data. Each data item has the format:

```
fieldname=fieldval
```

These items are separated by semicolon characters. The end of the record is a single newline. The get_record() function stores each fieldname, fieldvalue pair in the storage table pointed to by the *tp* argument. Use the functions in the `wordstore` package to store the data you read in. The function returns NO on end of file, and returns YES when after successfully storing a record.

The program should simply ignore blank lines and blank data items. The fieldname may be preceded by any number of space or tab characters; these characters are not part of the fieldname.

If get_record() encounters an error, it calls fatal(). There are two sorts of errors. The first sort is a

storage error, reported by the wordstore package. The second sort of error is bad input. If a given line does not look right, your function calls fatal(). 'Not looking right' means either (1) a field entry with a blank fieldname or (2) a field entry without an equals sign. Note that it is *not* an error to have an equals sign in a fieldval, nor is it an error to have a empty fieldval. It is also *not* an error to have spaces in the fieldname. Thus, this line:

```
name=Jane;  city=Boston;  ;;  college major=Chemistry
```

assigns values to three fields, "name", "city", and "college major", and contains two blank items, which are ignored.

You should read the fieldname and fieldvalue into arrays before passing them to the storage system. The fieldname may not exceed MAXFLD chars in length, and the value may not exceed MAXVAL chars in length. Any string that exceeds these limits will be truncated.

```
void mailmerge(symtab_t *tp, FILE *fmt)
```

mailmerge() prints out the report format, including data values at the appropriate spots. The function works as follows: mailmerge() copies characters from the format stream at *fmt to stdout inserting values for fields when it sees an insertion point. Insertion points are indicated in the format stream by *%fieldname%* . The values are stored in the symbol table passed as the *tp* argument.

If the record does not contain a field by that name, nothing is output there; no error is reported. If the user wants to place a real percent sign in the format file, it must be written as %% (as it is for printf).

Embedded Shell Commands

mailmerge() supports one addition feature: embedded shell commands. These items look like data insertion requests but do something quite different.

Say you wanted to include the current date as part of the report. You could add a "date=Nov 19, 2004" item to each data record and put the strings %date% in the template. But the date is not really part of the data, so it probably should not be added to that file. A better method is to run the Unix command date as part of the template. Similarly, if you wanted to include the computer running the fl program, you could run the hostname program as part of the template.

Running a Unix command from a C program is extremely easy. For example:

```
system("date");
```

runs the date command, and

```
system("ls -l $HOME")
```

lists the contents of your home directory.

Modify your program so that the special notation `%!unixcommand%` causes fl to run the specified unix-command when it encounters that field insertion point. Therefore, the fragment:

```
This letter was sent to you, %firstname% at %address% on %!date%
from a computer called %!hostname%.
```

will insert the firstname and address fields from the dataset and run the commands date and host-name.

Special Requirement for this project: For this project, just before your program calls the system() function, your program must call a wordstore13 function called table_export(tp) where tp is the symtab holding the current record. Doing so will allow the Unix command to refer to the values in the current record. Furthermore, your program should call fflush(stdout) before calling system() in order to make sure the output of the command appears at the correct place in the output.

`void clear_table(symtab_t *tp)`

clear_table() deletes all the fieldname, fieldvalue pairs stored in the wordstore table passed as an argument. You must add this function to the wordstore13.c file. This will add one more function to the package. It calls the library function free() for each string and struct in the list.

Hints, etc

Think about the overall structure of the project before you start coding. What functional units does it require, and how will you organize those into files. You already have a storage system; what other systems do you need to add to it?

When reading data, do not read entire lines. Users may put many fields per record, and using fgets() limits the line length to the size of the buffer. It is better to write a function that reads in one field at a time. The program imposes limits on the length of each fieldname and each value, so this field-reading function can impose those limits without restricting the total length of a record.

Get the mailmerge() function working for the fields from the data file. Once that is working, think about how to add in the embedded shell command virtual field.

Other than the global variable for the `fatal()` function, do not use any global variables in your code. Each part of the program operates as a separate machine. This makes it easy to expand and safe to modify. If you pass information through special global variables, you will compromise this modularity.

Your mailmerge function will need to 'rewind' the format file each time it re-reads it. This is simply done with **fseek(fp, 0L, SEEK_SET);** which resets the file pointer so it reads from the start of the file.

Write short functions. Anything over about 30 lines is too long. Make each function perform a well-defined operation. Some of the design points are based on your writing short, clear functions.

5. Get this Part Working First

Set up a directory for the project, cd into it, then type

```
cp ~lib113/hw/fl/files/*  .
```

In each of your files, #include "fl.h" and "ws13.h" to make sure everyone agrees on the definitions. Compile your files and fl.c and process.c , then link them. Test them with

```
fl sample.fmt < sample.dat
```

Modify the sample files to see how well your program works.

6. Part II - Adding Command Line File Names and Options

File Names So far, *fl* reads data from standard input. That means you type the data directly or redirect it from a data file. Once you have this version working, modify main() so it allows the user to type:

```
% fl sample.fmt datafile1 datafile2 ...
```

If no data files are specified on the command line, the program should read data from stdin.

Options In addition to accepting names of data files on the command line, your program should also accept command line options to specify the field separator and the record separator. The default field separator is semicolon, but the -d*X* option changes it to *X*. Making the newline char part of a command line option is tricky, so your program must also support the -D option which sets the field delimiter to '\n'. The default record terminator is newline, but the -r*Y* option changes it to *Y*. *Note:* These options make it possible to change the field separator to a newline and the record separator to another character.

With this change in place, your program should accept

```
% fl -d, sample.fmt datafile1 -r# datafile2
```

These options may appear anywhere in the line and affect the data files that follow the options. In fact, the separators can be set to different values for different files. This command:

```
% fl sample.fmt -d, datafile1 -d/ datafile2
```

sets the field separator to ',' for datafile1 and to '/' for datafile2. Which functions do you need to modify? Do not use global variables to handle this feature. Instead, modify the function arguments to pass the additional information.

7. Part III - Preparation, Makefile, and gcc -Wall

We provide some files of sample data, but you will understand the problem more clearly if you compose some test data of your own. By making up sample format files and sample data files, you can think through what processing your program has to do. For this assignment, you must turn in two sets of test data -- that is two format files and their corresponding data files.

Your program will be composed of a collection of files, a good reason to use the Unix `make` utility. For this assignment, you must create a makefile that can be used to build the entire program.

You also have had enough programming experience so you should be able to write code that will pass 'gcc -Wall' without complaints. Your code must do so.

8. Wrapping Up

You may test your program by running:

```
% ~lib113/hw/fl/test.fl
```

This shell script will test various aspects of your program and will report which parts do not work correctly.

Use the command `~lib113/handin fl` to turn in electronic copies of:

1. Full source code including a Makefile
2. Your test data and a sample run with your test data
3. A run with the our sample data
4. A typescript of the program being run by the test.fl script
5. A typescript of the program being compiled with gcc -Wall