

CS 1723, Data Structures
Spring Semester, 1997
Programming Assignment 4
Finding Words in a String
Due February 28, 1997

The Assignment: Start with a reference string **ref** of letters and an on-line dictionary of words. For example, **ref** might be **"horseshouses"**. You are to write a program which will find all dictionary words that can be made up from letters in the reference string. Notice that you can only use as many of each kind of letter as actually appear in the reference string: in the example above, up to 2 **"o"**s, 4 **"s"**s, 2 **"e"**s, and 2 **"h"**s, but only one **"u"** and one **"r"**. Thus your program must *not* find a word like **"usurer"** that has 2 **"u"**s in it or like **"sorrow"** that has 2 **"r"**s in it. Using the supplied large dictionary, over 50 entries will be found from this particular reference string, including **"horseshoe"**, **"osseous"**, and **"rhesus"**.

The Program: You may use a single file for your program. Make a new directory, **assign4**, and store the source file there as **findwords.c**. The organization of **findwords.c** should be similar to the following steps:

1. Obtain the name of the dictionary file from a command line argument.
2. Open the dictionary file, for reading. Open a file for writing to store the words found.
3. Prompt the user for the reference string, **ref**.
4. Read the reference string.
5. Read each line of the dictionary file, from the file opened for reading in step 2.
6. Check if each dictionary entry can be made up from the letters of **ref**.
7. If it can be made up from **ref**, insert this entry into a growing array of pointers to strings, **tab**.
8. Each entry found should also be written to a file opened for writing in step 2.
9. At end-of-file on the dictionary file, inform the user of the number of words found that can be made up from **ref**. Query the user if the words should be displayed. If **"yes"**, then print the contents of the array of pointers.
10. Close the two files from step 2.

Details:

1. Use the following format for a command:

```
runner% findwords -d words
```

Your text (the white book) tells how to extract these arguments (Section 5.10, pages 114 on). **argv[0]**, which is **"findwords"** and **argv[1]**, which is **"-d"**, can be ignored, but **argv[2]** should be the name of the dictionary file.

2. The actual 50 000 word dictionary (200K bytes) is

```
~wagner/pub/CS1723/words
```

You *must not* make a copy of this file, but instead *must* set up a *symbolic link* to it with the command:

```
runner% ln -s ~wagner/pub/CS1723/words words
```

The command should be executed inside your **assign3** directory. Afterwards any reference to **"words"** in this directory is translated to a reference to my dictionary. In case of a mistake, you can remove the symbolic link with a **rm** command, just as you would remove an ordinary file. (Of course, this will only remove the *link* and not the file linked to.)

The file opening will *not* work with

```
"~wagner/pub/CS1723/words"
```

in the **fopen()** function. The full path name

```
"/home/faculty/wagner/pub/CS1723/words"
```

will work, but I want you to practice using symbolic links.

The dictionary itself is just a list of words, and includes entries that are not normally considered English words, such as an entry for each single letter, and entries which are acronyms.

The **FILE** declarations, and the opening of these two files should be handled with **fopen()**, as described in the white book (Section 7.5, page 160 and Section B1.1, page 242).

3. To "prompt the user" means to write a short message to **stdout**.
4. Read the characters of the reference string **ref** from **stdin**.
5. I suggest that you use **fgetc()** to read a line (= record or entry) of the dictionary, character-at-a-time, until the newline. You must insert the null character yourself. You could mimic the white book's **getline()** function (page 29), except that you are not reading from **stdin**, and you do *not* want to insert a newline into the string you are reading (as **getline()** does).
6. This is actually fairly hard to think through. There are many methods. Don't worry about efficiency. We will discuss several approaches in class.
7. Once you have identified a dictionary entry that can be made up from the letters of the reference string, you are to insert this new string into **tab**, an array of pointers to **char**. The white book, pages 108-109, shows this, though you should use **malloc()** to allocate storage for each string.
8. I suggest that you write character-at-a-time, using **fputc()**. Don't forget to write the newline.
9. Page 109, bottom, shows one way to write this out.

Debugging Notes: Of course you should use a **makefile** and **lint**, as we have been doing.

To keep from going crazy with this assignment, you should get the pieces working *separately*. Try out the fetch of the third command line argument. Create a short test dictionary **"testwords"** to use, and first just try opening it and reading from it. Then try storing its entries into an array of pointers to **char**. Separately, exercise your function that checks if a given word can be made up from the characters in the reference string.

When you open a file with **fopen()**, a **NULL** pointer is returned if there is any trouble. The failure of an open is common, since the slightest mistake in the file name will cause failure. So you should *always* check for this null pointer.

Required Execution: For full credit it will be enough to execute this program with **"horseshouses"** as the reference string. Again for full credit your program should have all the various features mentioned in items 1 through 10 above. Some of these I will just have to check by looking at the code.