

CSc 352 (Fall 2018): Assignment 6

Due Date: 11:59PM Sat, Oct 20

The purpose of this assignment is to do more involved work with pointers, linked lists, and memory allocation.

General Requirements

1. Your C code should adhere to the coding standards for this class as listed in the Documents section on the Resources tab for Piazza. This includes protecting against buffer overflows whenever you read strings.
2. Your programs should indicate if they executed without any problems via their *exit status*, i.e., the value returned by the program when it terminates:

Execution	Exit Status
Normal, no problems	0
Error or problem encountered	1

3. Under *bash* you can check the exit status of a command or program *cmd* by typing the command "**echo \$?**" immediately after the execution of *cmd*. A program can exit with status *n* by executing "**exit(*n*)**" anywhere in the program, or by having **main()** execute the statement "**return(*n*)**".
4. Remember your code will be graded on lectura using a grading script. You should test your code on lectura using the **diff** command to compare your output to that of the example executable.
5. To get full points your code should compile without warnings or errors when the **-Wall** flag is set in **gcc**
6. Anytime you input a string you must protect against a buffer overflow. Review slides 82 – 87 of the basic_C deck.
7. You must check the return values to system calls that might fail due to not being able to allocate memory. (e.g. Check that malloc/calloc don't return NULL) getline() is an exception to this rule.
8. Your code must run without errors using valgrind. NOTE you do not need to free memory. Valgrind might report things called memory leaks. Don't worry about this. What you need is to see the line that includes: "ERROR SUMMARY: 0 errors"

Testing

Example executables of the programs will be made available. You should copy and run these programs on lectura to test your program's output and to answer questions you might have about how the program is supposed to operate. Our class has a home directory on lectura which is:

`/home/cs352/fall118`

You all have access to this directory. The example programs will always be in the appropriate **`assignments/assg#/prob#`** subdirectory of this directory. They will have the same name as the assigned program with "ex" added to the start and the capitalization changed to maintain camelback. So, for example, if the assigned program is **`theBigProgram`**, then the example executable will be named **`exTheBigProgram`**. You should use the appropriate UNIX commands to copy these executables to your own directory.

Your programs will be graded by a script. This will include a timeout for all test cases. There must be a timeout or programs that don't terminate will cause the grading script to never finish. This time out will never be less than 10 times the time it takes the example executable to complete with that test input and will usually be much longer than that. If your program takes an exceedingly long time to complete compared to the example code, you may want to think about how to clean up your implementation.

Makefiles

You will be required to include a Makefile with each program. Running the command:

`make progName`

should create the executable file *progName*, where *progName* is the program name listed for the problem. The gcc command in your Makefile must include the **`-Wall`** flag. Other than that, the command may have any flags you desire.

Submission Instructions

Your solutions are to be turned in on the host **`lectura.cs.arizona.edu`**. Since the assignment will be graded by a script, it is important you have the directory structure and the names of the files exact. Remember that UNIX is case sensitive, so make sure the capitalization is also correct. For all our assignments the directory structure should be as follows: The root directory will be named **`assg#`**, where # is the number of the current assignment. Inside that directory should be a subdirectory for each problem. These directories will be named **`prob#`** where # is the number of the problem within the assignment. Inside these directories should be any files required by the problem descriptions. For this assignment the directory structure should look like:

```
assg6
  probl
    noVowels2.c
    Makefile
```

To submit your solutions, go to the directory containing your assg6 directory and use the following command:

```
turnin cs352f18-assg6 assg6
```

Problems

prob1: noVowels2

Recall in assignment 2 we defined two words are *noVowel matches* if the words are the same when we remove their vowels (a, e, i, o, u). This problem involves writing code to divide up a set of words into groups where each group is a set of words that are noVowel matches of each other.

Write a C program in a file called **noVowels2.c** and a **Makefile** that creates an executable called **noVowels2** to classify words as noVowel matches as specified below.

- **Input:**

The input comes from **stdin** and consists of a sequence of strings separated by white space. This means the strings may be on one line or on multiple lines.

- **Behavior:**

Your program should read in a sequence of words from the input stream, group them into sets where each set consists of words that are noVowel matches of each other, and print out the resulting words one group per line as specified below.

Multiple occurrences of a word in the input should be retained, i.e., the total number of legal words in the output should be the same as the total number of words in the input.

For the purposes of this program, a *word* is a sequence of alphabetic characters. The property of being a noVowel match is case-insensitive.

Note that it is possible if a word is all vowels. In this case your program should ignore this word. It is not an error, but it should not appear in the list of groupings.

- **Output:**

The output from your program should be printed out as follows (see example below):

- each group of noVowel matches should be printed out on a separate line;
- The order of the noVowel match groups should be the same as the order they appear in the input. In other words, if *dog* is the first word input, then the first line of the output should be all the noVowel matches of dog that appear in the input. So each line will include all the words that are noVowel matches of each other. The order of the words per line should also be the order in which they appear in the input. Also, although case should not be considered when determining if words are noVowel matches of each other, when printing out the words the original case (the case of the words in the input) should be preserved. In other

words, if a word appears as **WoRd** in the input, it should be printed out as **WoRd**.

- **Assumptions:**

You may assume that each word has length at most 64, though your program should still protect against buffer overflow if a word longer than 64 chars is entered.

- **Error Conditions:**

A string that contains something other than an alphabetic character is an error. When such strings are found in the input, an error message should be printed, the string should be ignored, and the next string would then be read in.

- **Restrictions:**

The point of this problem is to get experience using `malloc()` or `calloc()` to create linked structures. Therefore you may NOT use an array where you allocate some large amount of space for your list of words, perhaps reallocating more space as necessary. Instead you should use a linked list or similar structure that uses exactly the amount of memory you need. I'm not saying you can't use arrays in your assignment. You certainly will need an array to fit the longest possible word to use with `scanf()`. You are allowed to use static arrays in the program as you need them, but you must use a linked list to store your list of noVowel equivalence groups and the list of words within those groups. See the comment section below.

Also, rather than include in your structs a large array to save the largest possible word, include an `char *` and allocate exactly the amount of memory you need. The Arrays, Ptrs, Structs deck has an example program that reads words and stores them in a linked list. Look over that example if you're confused.

- **Makefile:**

In addition to your source files, you should submit a make file named **Makefile** that supports at least the following functionality:

make noVowels2

Compiles the C source code to create an executable named `noVowels2`. The compiler options used should include `-Wall`.

- **Example:**

Given the list of words

```
Hat  mast  past  hit  mostly  Past
hoot  prove  Pest  HATE  prove  paste  most
```

the output generated should be as follows:

```
Hat hit hoot HATE
mast most
past Past Pest paste
mostly
prove prove
```

- **Comment:**

A good data structure for this program is a linked list for the group of noVowel matches, each node of which contains a linked list of the strings for each of those matches. In other words, use two different node types. One for the set of matches, and another to hold the actual words input.