

CSc 352 (Fall 25): Assignment 2

Due Date: 11:59PM Friday, Sep 19

The purpose of this assignment is to work with strings and arrays, and to get acquainted with various string library routines. You should use **scanf** to read in the strings. We've seen examples in class on how to do that.

General Requirements

1. Your C code should adhere to the coding standards for this class as listed in the Documents section on the Content tab for D2L.
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 71 – 76 of the basic_C deck. For this assignment what that means is you should never have a statement like **scanf("%s", str)** but instead use something like **scanf("%16s", str)** where the number (in this case 16) is one less than the size of the char array **str**. (In this case **str** would be declared by: **char str[17];**)

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/fall125

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.

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:

```
assg2
  prob1
    anagrams.c
  prob2
    baseChanger.c
```

To submit your solutions, go to the directory containing your assg2 directory and use the following command:

```
turnin cs352f25-assg2 assg2
```

prob1 anagrams

Write a C program, in a file **anagrams.c**, that identifies and prints out anagrams from a set of strings, as specified below.

(A string A is defined to be an anagram of a string B if we can construct B simply by rearranging the characters in A . It follows that (1) every string is an anagram of itself; and (2) if A is an anagram of B , then B is an anagram of A .)

- **Program behavior:** Your program should read a sequence of strings from **stdin** until no more strings can be read. For each string S so read, if S is an anagram of the first string in the input sequence, then S is printed out on **stdout** according to the format specified below.

Since the first string is necessarily an anagram of itself, it is always printed out.

If **stdin** does not contain any strings to be read in (i.e., the input routine encounters **EOF** right away), your program terminates normally. (**Note:** this affects the value that the program returns.)

- **Input format:** Each input string consists of a sequence of alphabetical characters (i.e., upper case letters and lower case letters). If an input string (other than the first one) contains any non-alphabetical character, you should give an error message, ignore the offending string, and continue processing the rest of the input. If the first string contains a non-alphabetical character, then you should give an error message and exit the program.

You may assume that each input string is at most 64 characters long (although you must still protect against buffer overflow).

Note: There is a C library function that will identify alphabetic characters which you are welcome to use if you wish.

- **Output format:** Each string S that is an anagram of the first string should be printed out using the statement

```
printf("%s\n", S);
```

- **Case sensitivity:** The property of one string being an anagram of another is not case sensitive. Thus, **taper** and **Pater** are anagrams of each other. However, the strings you print out must be the same as the strings you read in. In other words, any processing you do to ignore cases should not affect the strings themselves.

Note: There are C library functions **toupper()** and **tolower()** that can be used for case conversions; use **man** for details. I recommend experimenting with these functions in some small toy programs to get a sense of their functionality.

- **Example:** Suppose that the input to the program is the following (in reality they will be read from **stdin** so they may or may not all be on one line):

Star torus Rats tarsus star roust tsrA

The output should be

Star
Rats
star
tsrA

Note that although **torus** and **roust** are anagrams of each other, they are not printed out because they are not anagrams of the first string in the input, i.e., **Star**.

prob2: baseChanger

Numbers can be written using any base system. In computer science we are used to binary (base 2), octal (base 8), or hex (base 16) and well as normal decimal (base 10). But the idea of a base can extend to any number. This problem requires you to write code that will read in a base, followed by a series of white space separated strings that represent numbers in that base. For each of these strings, convert it into an unsigned long and print the value out using `printf("%lu\n")`. The “%lu” is the code for unsigned long.

We will limit ourselves to using alpha numeric symbols (e.g. 0, 1, ... , 9, a, b, c, ... , z). Thus the largest base we will allow is 36. Base 1 numbers are kind of a special case, so legal bases will be 2-36. If a base outside that range is entered, the program should exit with an error.

Since the digits take on values from 0 to 9, the letters (if needed) will take on the values **a**=10, **b**=11, ... , **z**=35.

Thus suppose the base 13 is entered followed by a84b. (For a base 13 number only the letters **a**, **b** and **c** are legal)

The conversion would be $(11 + 4*13 + 8 * 13^2 + 10*13^3) = 23385$

(Hint: While this is the formula for the conversion, you don't want to implement this formula directly. Use [Horner's rule](#) for evaluating polynomials we will talk about this in class on Tuesday, Sep 17)

As you might guess, these numbers might get very big, which is why we want you to use unsigned long numbers for your calculations. You can print an unsigned long using printf with the code “%lu” .

Our number system will not be case-sensitive, so **A84B** will also evaluate to **23385**.

Your source file should be called **baseChanger.c**

Error conditions:

If the first thing entered (use **scanf** to input an integer) is not an integer between 2 and 36 inclusive, then you should print an error message to **stderr** and exit with the appropriate code.

Suppose the base **b** has been entered. You will then read in the following b base numbers (use **scanf** to read a string) one at a time. If the string is not a legal b-base number (e.g. it contains a non-alphanumeric character or a letter or digit too big for the base), an error should be printed out (always ALL errors go to **stderr** for all programs we write) and then next b-base number

read in. You may use a library function to check for alphanumeric characters. You may also use a library function to change the case of letters if you wish.

Note you may **assume** that all b-base numbers entered are no more than 6 characters long. By *assume* I mean you don't have to check for the length, but you still should protect against buffer overflow.

Restrictions:

1. A simple, but very tedious way to write this program would be to use a long series of if-then-else statements, or a switch statement with one case per each letter, to get the value of each letter. For example

```
if (str[i] == 'a') {  
    num = 10;  
} else if (str[i] == 'b') {  
    num = 11;  
}  
etc.
```

This would be ugly. We don't do ugly. Therefore, such solutions are banned. You have to figure out a different way to get the value the letters. (Hint: think about what we did in class to convert a string to lower case.) Note this does NOT mean you can't use if's or switches in your program. You will need if's. In fact, since the ASCII encoding for the digits is not consecutive for the ASCII encoding for the letters, you will need to break the conversion of a single letter to a number into two separate cases.

2. You are not allowed to use any functions from the math.h library for this problem.

Students want to use pow(), but this is inefficient, clumsy, and unnecessary. We will talk about this in class, but you can get the algorithm from [Horner's rule](#) if you want to work it out before then.