CSc 352 (Fall 2018): Assignment 2

Due Date: 11:59PM Saturday, Sep 15

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 Resources tab for Piazza.
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. 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/fall18`

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
      mayan.c
   prob2
      palindromes.c
   prob3
      noVowels.c
```

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

turnin   cs352f18-assg2   assg2

Problems

prob1 mayan

The Mayans used a base-20 number system. This problem requires you to write code that will read in a base-20 representation of a number and write it out as a base-10 (decimal) number.

For the purposes of this problem, base-20 numerals will be as follows:

a = 0; **b** = 1; **c** = 2; ... **t** = 19.
For example, the base-20 number "**hmmm**" evaluates as follows. Since **h** = 7 and **m** = 12, we have
hmmm = $7 \times 20^3 + 12 \times 20^2 + 12 \times 20^1 + 12 \times 20^0$ = **61052**.
Our number system will not be case-sensitive, so **HmMm** will also evaluate to **61052**.

Write a program, in a file **mayan.c**, that reads in Mayan numbers using the notation described above, and writes out the corresponding decimal values to **stdout**. Your program should iteratively read in Mayan numbers and print out the corresponding decimal values until EOF is encountered.

Error conditions: Your program should report an error if the string read in contains any character that is not a valid base-20 numeral. In this case, the string being processed should be discarded and processing should continue. Error messages should be reasonably informative, and should be sent to `stderr`.

Output: For each mayan number read, the decimal value of that number, **val**, should be printed using the statement

```
printf("%d\n", val)
```

Assumptions: To simplify programming, you may assume that:

1. each base-20 number is at most 6 characters long; and
2. there are no negative or fractional values.

When I say you may assume something, I mean we won't have any test cases that violate this. Thus, we will not have any test cases with strings longer than 6 characters. It doesn't matter what you program does in these situations since it will not be tested. Note, however, that you will still want to protect against a buffer overflow when reading the string in. (See section on general requirements.)

Restrictions:

1. An obvious 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 base-20 numeral, to get the value of a base-20 numeral, e.g., something like:

```
if (ch == 'a') val = 0;
else if (ch == 'b') val = 1;
...
else if (ch == 't') val = 19;
else ...    /* report error */
```

OR

```
switch {
case 'a': val = 0; break;
...
case 't': val = 19; break;
default: ...     /* report error */
}
```

   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 of a base-20 numeral. (**Hint:** if you look at the ASCII character code you may notice that the ASCII encodings of the lower-case base-20 numerals form a range of values without any gaps, and similarly for the upper-case numerals. How can you take advantage of this?)

2. Your solution should not use math library functions such as `pow()`. These are not needed for this program, and would only make the code more complicated than necessary. Instead, consider using Horner's rule for evaluating polynomials.

prob2: palindromes

A *palindrome* is a string that reads the same forwards and backwards. For example, *mom* and *toot* are palindromes, but *mummy* and *trust* are not.

This problem involves writing code to recognize palindromes. Write a program, in a file **palindromes.c** that behaves as specified below.

- **Definitions:** For the purposes of this assignment, a *word* is any sequence of non-whitespace characters. For example, **uncle** and **cl3v3r** are words, as are **011010010** and **abc$!^def**. However, **abc def** is not a word (it contains a whitespace character).

  This definition implies that a word can be read in using `scanf("%s" ...)`.

- **File names:** Your source code should be in a file named **palindromes.c**.

- **Behavior:** Your program should repeatedly read in words from the input until no more words can be read in. For each word read in, your program should determine whether or not that word is a palindrome. Your program should print out the value **1** on **stdout** if the word is a palindrome, and **0** if it is not, as described under "**Output**" below.

  You should not distinguish between upper and lower case letters for this problem, i.e., the word *Mom* should be considered to be a palindrome. The simplest way to deal with this is to convert all of the letters in the input word to the same case, either upper or lower, before processing it further: see the library functions `toupper()` and `tolower()`.

  There are no error cases for this program. Your program's exit status should therefore be **0**.

- **Output:** For each word, the value `val` indicating whether or not it is a palindrome should be printed using the statement

  ```
  printf("%d\n", val)
  ```

- **Assumptions:** You may assume that each input word is at most 64 characters long. (Though you still should protect against buffer overflows.)

- **Error cases:** None.

- **Example**: Given the input

  ```
  a
  an
  m0m
  muM
  tum
  xxx@xxx
  syzygy
  hah
  haha
  ```

  The output generated should be

  ```
  1
  0
  1
  1
  0
  1
  0
  1
  0
  ```

prob3: noVowels

There are written languages that don't use vowels. If we did that with English, how many words would no longer be distinguishable? Write a program, in a file **noVowels.c** that reads a series of strings, the first being the base pattern, and writes out the strings that are the same if the vowels are removed.

- **Definitions:**
    - For the purposes of this assignment, a *word* is any sequence of upper- and lower-case letters. For example, **uncle** and **clever** are words, but **abc$!^def** and **cl3v3r** are not. This definition implies that a word can be read in using `scanf("%s" ...)`, but requires additional checking to ensure that it is in fact a valid word.

    - A vowel is any letter in the set {a, e, i, o, u} or the uppercase version of these letters. Technically 'y' is sometimes used as a vowel. However, to keep things simple, for this problem vowels will only be the 5 letters mentioned.

    - We will say two words are noVowel matches of each other if they are identical (with the exception of case differences) when you remove their vowels. For example, **Dig** and **dog** are noVowel matches, but **did** and **dog** are not.

- **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 a noVowel match 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 matches 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.

  Note: There is a C library function which identifies alphabetic characters which you're welcome to use if you wish.

- **Output:** For each string S that is a noVowel match of the first string should be printed using the statement:

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

- **Case sensitivity:** The property of one string being noVowel match of another is not case-sensitive. Thus, **PITTER** and **patter** are noVowel matches 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.

- **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 standard stir stARE whip wehop
  ```

  The output should be

  ```
  Star
  stir
  stARE
  ```

  Note that although `whip` and `wehop` **are noVowel matches** of each other, they are not printed out because they are not noVowel matches of the first string in the input, i.e., `Star`.