Lab 2

Practice with strings

In this lab, we’ll practice working with C strings. Recall that these are just arrays of characters terminated with the character ‘\0’ (0 if the character is viewed as a number).

Begin by typing in the following program:

```c
#include <stdio.h>
#include <string.h>

int main() {
    char line[10];
    while(fgets(line, 10, stdin) != NULL) {
        printf("line read: %s\n", line);
    }
}
```

This program repeatedly reads lines from the keyboard (standard input, called stdin in the code) and prints them. The argument 10 to `fgets` is the length of the buffer (i.e. the array) that characters are being read into; this ensures that `fgets` doesn’t walk off the end of the buffer. With argument 10, `fgets` will read at most 9 characters, leaving room for the string-terminating 0.

Compile and run this program. Try it out by typing in short lines and seeing what it prints. Then type in a long string (more than 10 characters) and notice how it is split over multiple lines. This occurs because `fgets` returns after reading a buffer full of characters so your long line was read by multiple calls as separate lines. Exit the program by hitting control-D (C-D).

When you gave the program a long input line, you may have noticed that it behaves differently. For short inputs, there was a blank line after every printout, but there wasn’t one after partial lines. These blank lines are because `fgets` stores the newline character at the end of your line into the string it reads. Since our `printf` call also adds a newline, the result is a blank line.

To clean up the display, let’s remove the newline character from `line` before we print it. Add a loop that traverses `line` until coming across either a 0 (0 or ‘\0’) or a newline (‘\n’). If you encounter a newline, replace it with a 0, which has the effect of shortening the line by one character.

Run this program and verify that it no longer prints blank lines after the output.

Next increase the line length to 80, changing both the number of characters allocated for `line` and also the corresponding argument to `fgets`. Now we’re going to process the line looking for words so we want it to be long enough to have interesting words.

Begin by modifying the program to print just the first word. To do this, write a loop to replace the first space (‘ ‘) character in `line` with a 0. Check that your program only prints the first word.

Next, modify the program to print each word separately. To do this, you’ll want to find the endpoints of a word (which we’re defining as a sequence of non-space characters surrounded by either spaces or the string boundaries). Then copy these into a different array of characters (being sure to put in a string-terminating 0) and print that string.

Once you have a basic version of this working, be sure that it can handle strings with multiple spaces. (For example, “ hello there ” should only print “hello” and “there” even though there are many extra spaces.)
As a next step, see if you can standardize the spacing in a line; given an input line, change it so that every pair of words is separated by exactly one space and there are no leading spaces before the first word or trailing spaces after the last word. You can do this either by copying the words into a new string or by modifying the line in place.

If you finish those tasks, see if you can perform other interesting string functions. For example, can you make your program turn text into pig latin? (To convert a word into pig latin, follow the rules given on Wikipedia (https://en.wikipedia.org/wiki/Pig_Latin); move leading consonants to the end and add “ay”, words beginning with a vowel have “yay” added.)