Setting up Codio for this HW:

1) Open the Codio assignment via Coursera
2) From the Codio File-Tree click on: **program1.c**

Assignment Overview:

From lecture you have learned that in C an array of characters with a NULL termination is considered a string. An array of characters without a NULL termination is simply an array of characters. There is a library: <string.h> available with most C compilers (*sadly not the LC4 C-compiler*) that includes several functions designed to work with and manipulate strings in C. Strings are so common in C that knowing how to use these functions in C is considered a basic skill. The goal of this homework is to give you experience with strings in C, with the library of functions designed to work with strings, give you an appreciation of how those functions work, and also continue to help you work with pointers and arrays (in the context of C-strings).

Problem #1: Creating your own library of string functions

In lecture, we discussed a commonly used function: strlen() that is part of the <string.h> library. It's job is to take in a C-**string**, count the number of characters up to (but not including) the NULL character and return the string's length. As an example, if our string was:

```
char my_string [100] = "Tom" ;
```

strlen(my_string) would return 3. Even though there are 100 bytes allocated on the stack for the string, since there are only 3 characters (followed by a NULL), the length of the string is indeed 3.

In lecture, two versions of a strlen() like function were presented. One that uses array notation and one that uses pointer notation. Ultimately they perform the same function, but my_strlen() treats the incoming argument (char* string) as if it is an array and my_strlen2() treats the incoming argument as the pointer it truly is; using pointer arithmetic to determine the string's length.

```
size_t my_strlen(const char *str)
     int len = 0 ;
     while (str[len] != '\0') {
          len++ ;
     }
     return (len) ;
}

size_t my_strlen2(const char *str)
          const char* s;
          for (s = str; *s; ++s) ;
          return (s - str);
}
```

NOTE: `size_t` is not an actual C-type, is a "typedef'ed" shortcut for: **`unsigned long`**

For this problem, your job is to define your own library of string functions to mimic the standard C library string functions. On Codio, I have included a header filed called: **my_string.h**. In that header file I have declared several functions: my_strlen(), my_strcpy(), etc. In the file: **my_string.c**, I have defined only two of the many functions you must define yourself, namely: my_strlen() and my_strlen2(). In a third file: **program1.c**, I have provided some code that calls the functions in your "my_string" and compares the output of them to functions in the standard C-library: <strings.h>. Look carefully at these three files before continuing.

Your job is complete **my_string.c** (the implementation of the library) and **program1.c** (tests of the library's functions). You will notice that you must make two versions of the same function. As an example: my_strlen() uses array notation to solve a problem, while my_strlen2() uses a much more efficient implementation using pointer to solve the same problem. This is your chance to truly play with pointers and see if you can come up with more efficient techniques to solve problems that were done using array notation.

Since many of these functions will be new for you, I have included these helpful links so you can learn to use each of these functions before you try to implement them!

https://www.tutorialspoint.com/c_standard_library/c_function_strlen.htm
https://www.tutorialspoint.com/c_standard_library/c_function_strcpy.htm
https://www.tutorialspoint.com/c_standard_library/c_function_strchr.htm
https://www.tutorialspoint.com/c_standard_library/c_function_strcat.htm
https://www.tutorialspoint.com/c_standard_library/c_function_strcmp.htm

You may also be struggling with the "const" variable modifier, especially as it is used with pointers in C. The following tutorial is an excellent read:

http://www.geeksforgeeks.org/const-qualifier-in-c/

Makefile:

You will also need to create Makefile so that you can more easily compile your library and the code that test your library. Create a Makefile and implement the two directives:

```
my_string.o
```

```
program1
```

If you are struggling with Makefiles, the video tutorial provided before this assignment will be very helpful.

Problem #2: Adding non-standard string functions to your "my_string" library

In problem #1 you implemented and tested functions that mimic those in the standard C-library. Now you'll add some functions that don't exist in the C-library, but will exist in your own my_string library! The two functions are as follows:

my_strrev ()

The first function you'll implement will be called: my_strrev(). It should take in only 1 argument, a string. The function should reverse the contents of the string that is passed in. And similar to functions like strcat(), it should return a pointer to the resulting string. As an example:

```
char my_string [] = "Tom"
char* ptr = my_strrev (my_string) ;
```

After my_strrev() completes, `my_string`, should contain: "moT" and *`ptr`*, should point to the first element in my_string.

my_strccase ()

The second function you'll implement will be called: my_strccase(). It should take in only 1 argument, a string. The function should convert each character of the passed in string to the opposite case. As a hint, examine the ASCII table, you will see that if you work on the HEX characters directly, you can very easily convert them to their upper or lowercase equivalents!

Like my_strrev(), my_strccase () should return a pointer to the resulting string. As an example:

```
char my_string [] = "Tom"
char* ptr = my_strccase (my_string) ;
```

After my_strccase() completes, `my_string`, should contain: "tOM" and *`ptr`*, should point to the first element in my_string.

You do not need to make two versions of these functions (pointers and array). You can implement them anyway you see fit. You will need to modify my_string.h (adding the proper declaration statement each of these new functions) and my_string.c (adding the proper definition of each of these new functions). And you will need to create **program2.c** to properly test these new function's you've created.

Makefile:

Be sure to add a "program2" directive to your existing Makefile.

Problem #3: Parsing Strings

There are two very useful functions, related to scanf() and printf() that are frequently used to parse strings and convert data from strings into different data types. These two functions are "sscanf()" and "sprintf()". They work identically to scanf() and printf(), except instead of reading the keyboard for input, sscanf() reads a "string" as its input. And instead of using the ascii display for output, sprint() uses a string as its output. Since you already know how to use scanf() and printf(), this problem will feel familiar. These two links will provide a reference for their arguments/returns and basic function:

https://www.tutorialspoint.com/c_standard_library/c_function_sscanf.htm
https://www.tutorialspoint.com/c_standard_library/c_function_sprintf.htm

arguments to main()

Recall from our lectures on the stack, that the function main() always has a blank spot for arguments, but our declaration of main() never has any arguments. Well, it's actually possible for main() to take arguments. BUT, they can only be specified as follows:

```
int main (int argc, char** argv) ;
```

The first argument: "argc" contains the # of arguments someone has passed to main. The second argument: "argv" is actually a pointer to an array of strings. Your next question should be, exactly how do you pass arguments to main()? You do it when you start your program in the terminal. Take the following code and put it in a file called: program3.c

```
int main (int argc, char** argv) {

     printf ("# of arguments passed: %d\n", argc) ;

     for (int i=0; i< argc ; i++) {
          printf ( "argv[%d] = %s\n", i, argv[i] ) ;
     }

     return (0) ;
}
```

Add a directive to your Makefile to compile this program as object file "problem3". Next, run as follows:

```
./program3 arg1 2 arg3 4 arg5
```

Watch the output and look at the code above to see how it works! Try it with different arguments and watch how things change.

The problem to solve:

Notice that all the arguments passed in are treated as strings in your program. Even though "2" is a number, it is treated as a NULL terminated character array (a string) inside the argv array. Your job is to add to the above program to make it do the following:

1) convert any argument that is actually a number, from its string form into an integer using sscanf(). *As a small hint, look at the return type of sscanf(), notice that if it makes a match, you get the # of matches it made to your string as a return.*

2) store any integer arguments into an array of integers (you may assume we'll never pass in 10 integers at any time). For the above example (`./program3 arg1 2 arg3 4 arg5`), your program would generate an array: `{2, 4}`

3) store any non-integer argument (example: arg1) into 1 large string – separated by spaces, using sprint(). For the above example, your program would generate a string: `"program3 arg1 arg3"`. You may assume a maximum length of 250 for this string.

4) print out the contents of your integer array (a newline after each element) and the contents of your single string.

Extra Credit Problem #4 (6 points): Adding non-standard string functions to your "my_string" library

One of the more difficult string functions to use and to implement is "strtok()". If you would like a chance at extra credit, implement this function as well in your "my_string" library: https://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm

You must test it out in a file called: program4.c. You must also update your Makefile as well.

Important Notes on Plagiarism:

- Nearly all of the functions you are asked to write in this HW are available online! If you steal the code, copy and paste it, and submit it as your own, you will have learned nothing. You will also get caught, as we have all the standard versions of these functions in our plagiarism detector!
- We will scan your HW files for plagiarism using an automatic plagiarism detection tool.
- If you are unaware of the plagiarism policy, make certain to check the syllabus to see the possible repercussions of submitting plagiarized work (or letting someone submit yours).