COMSC-165 Lab 7
C Strings and C++ Strings

In this lab session you will exercise **`char`** arrays, which are used in C to simulate the C string data type. You will write your own `int main` to test your functions. You will also get to write your own code, based on specifications provided.

This is particularly important, because we will be using C strings for our work through most of the remainder of Comsc-165.

Note that a **`char`** array is not necessarily a C string -- C strings are **`char`** arrays that include a null terminator to mark the end of the C string. Also note that a **`char`** array used to contain a C string *must* include enough elements for all the characters in the C string, *plus* its null terminator. E.g., to store the 12-character C string "Hello, World", which has 10 letters, one comma, and one space, a **`char`** array must be at least 13 elements in size.

As you complete this assignment, post the required file(s) to the COMSC server. For assignments that involve more than one file, you may post them all at once, or one-at-a-time as you complete them, as you prefer. For assignments that involve more than one separate "lab" (like lab 1a, 1b, and 1c), the individual labs are graded in order, starting with "a", which must be fully correct before "b" is graded, and so on. This assignment is worth 50-points, to be awarded after all labs are successfully completed. Use the "Submit your FA2014 work" link on the class website to post your file for this lab, using the **lab7** folder provided.

Note: You will write functions in each of the first four steps. These functions are not supposed to "know" the size of the char arrays passed to them in their parameter lists -- they must assume that the char arrays contain sentinels to mark their ends. Do not use global variables or constants in any way. Do not write to `cout` in the functions. No points will be awarded for functions that use any other method than sentinel markers to deal with the sizes of arrays.

LAB 7: Working With C Strings [`Strings.cpp`]
Create a console application named **Strings.cpp**. Do NOT submit until you complete the last step.

STEP 1 of 7: Write A C-String Length Function
C++ has one, but C has no "string" data type. So we use arrays of characters to achieve the same functionality. Write a C function to *compute C string length*, using this function *prototype*:

`int myStrLen(const char*);` *// return length of null-terminated C string stored in char array*

Refer to your notes from our lectures on arrays, for how to write this function. Remember to search for the null character, **`'\0'`**, to define the end of a C string -- i.e., the character in the array after the last character of the C string. Test for it using, e.g., **`if (array[i] == 0)`**, or **`if (array[i] == '\0')`**.

Do *not* use pointers -- use array syntax instead.

Do *not* use more than one loop in the function.

Be sure to fully test the function by putting several calls in a **`main`**function, using a variety of different C

strings. Name the CPP file as you wish -- it is not until step 7 that you will be asked to submit this function in a program named **Strings.cpp**.

Do not use **<cstring>***'s* **`strlen()`**, **`strcmp()`**, *or* **`strcpy()`** *functions in your program -- you are to write your function using loops to process the* **`char`** *arrays.*

STEP 2 of 7: Write A C-String Copy Function
Write a C function to *copy one C string into another*, using this function *prototype*:

```
void myStrCpy(char*, const char*); // copy the contents of the 2nd C string into the 1st
```

Refer to your notes from our lectures on arrays, for how to write this function. Remember to search for the null character, **`'\0'`**, to define the end of a C string -- i.e., the character in the array after the last character of the C string. Test for it using, e.g., **`if (array[i] == 0)`**, or **`if (array[i] == '\0')`**. *Do not use* **<cstring>***'s* **`strlen()`**, **`strcmp()`**, *or* **`strcpy()`** *functions in your program -- you are to write your function using loops to process the* **`char`** *arrays.*

Do *not* use pointers -- use array syntax instead.

Do *not* use more than one loop in the function.

STEP 3 of 7: Write A C-String Compare Function
Write a C function to *compare a C string to another for equality of content*, using this function *prototype*:

```
int myStrCmp(const char*, const char*); // return 0 if two C strings are equal, else return 1
```

Refer to your notes from our lectures on arrays, for how to write this function. Remember to search for the null character, **`'\0'`**, to define the end of a C string -- i.e., the character in the array after the last character of the C string. Test for it using, e.g., **`if (array[i] == 0)`**, or **`if (array[i] == '\0')`**.

Do not use **<cstring>**'s **`strlen()`**, **`strcmp()`**, or **`strcpy()`** functions in your program -- you are to write your function using loops to process the **`char`** arrays. Do *not* use pointers -- use array syntax instead. Do *not* use more than one loop in the function.

STEP 4 of 7: Write A C-String Swap Function
Write a C function to *swap the contents of two C strings*, using this function *prototype*:

```
void myStrSwap(char*, char*);
```

Swap each `char` value in the parameter C strings until the null terminator of the *longer* string is swapped. *Do not use* **<cstring>***'s* **`strlen()`**, **`strcmp()`**, *or* **`strcpy()`** *functions in your program -- you are to write your function using loops to process the* **`char`** *arrays.* Do *not* use pointers -- use array syntax instead. Do *not* use more than one loop in the function.

STEP 5 of 7: Write A C-String Uppercase Conversion Function
Write a C function to *convert a C string to uppercase*, using this function *prototype*:

```
void myStrUpr(char*);
```

Do not use **<cstring>***'s* **`strlen()`**, **`strcmp()`**, *or* **`strcpy()`** *functions in your program -- you are to write your function using loops to process the* **`char`** *arrays.* Do *not* use pointers -- use array syntax instead. Do

not use more than one loop in the function.

STEP 6 of 7: Write A C-String Lowercase Conversion Function
Write a C function to *convert a C string to lowercase*, using this function *prototype*:

```
void myStrLwr(char*);
```

Do not use ***<cstring>****'s* ***strlen()****,* ***strcmp()****, or* ***strcpy()*** *functions in your program -- you are to write your function using loops to process the* ***char*** *arrays.* Do *not* use pointers -- use array syntax instead. Do *not* use more than one loop in the function.

STEP 7 of 7: Combine All Functions Into A Test Program Create a new console application named **Strings.cpp**. Include in it the functions you developed and tested in step 1-6 above: **myStrLen**, **myStrCmp**, **myStrCpy**, **myStrSwap**, **myStrUpr**, and **myStrLwr**. To do so, simply copy and paste them into **Strings.cpp**.

Include enough testing to convince yourself that the functions will work right in *any* C/C++ program that uses them. You can assume that the user of your functions will properly size their C strings, and include null terminators to mark the ends of strings. Your program will *not* be scored using *your* main!

Post ***Strings.cpp*** *to the COMSC server.*

GRADING POLICY
Lab work will be returned for you to fix and resubmit for any of the following reasons, or for not following stated specifications, unless directed otherwise by your instructor, Point penalties will be assessed per the syllabus for each time a lab is returned for redo for any listed reason below or specification above. Points will be awarded for an assignment upon final, successful completion of all of its labs. Note that you will NOT receive a list of corrections to make, because to do so would remove any incentive for students to do quality work. Instead, grading of your work will stop upon discovery of the first mistake. Grading will not continue until the mistake is corrected. So to avoid multiple cycles of correction and resubmission, and the cumulative point penalties, make sure that your work is as correct as you can make it before submitting it.

Basic checklist, even if your compiler "forgives" doing otherwise:

1. Spell and case filenames *exactly* as specified. To avoid errors, be sure to configure your Windows computer to NOT hide file name extensions -- refer to the Burns Intro to Programming text, chapter 2.
2. Submit files to the specified folder. Do NOT submit archive files, like ZIP, JAR, or RAR.
3. Include identifying information as the first lines in the file AND as the first lines of your output in main.
4. Do NOT use the statement `system("PAUSE");` or any other system-specific code.
5. Do NOT try to access the value stored in an uninitialized variable.
6. CPP and H files must be free of careless typing mistakes that would prevent compilation.
7. Put an `endl` as the last item in the last-executed `cout` statement before the closing curly brace of `main`.
8. Do NOT forget to use the `cin.ignore` statements for non-string console input, unless specifically directed otherwise.
9. Do NOT forget to use the `fin.ignore` statements for non-string file input, unless specifically directed otherwise.
10. For file I/O, NEVER include a path designation. ALWAYS use the "working folder" for files.

11. Use the proper `#includes` and `usings` as listed in the C++ Library PDF.
12. Do NOT #include C-libraries such as `math.h` -- include their C++ equivalents instead, like `cmath`.
13. Do NOT #include libraries that you do not use.
14. Do NOT #include "**stdafx.h**"; do NOT use `pragma`.
15. Do NOT use C/C++ constructs or library items that are not specifically taught in this class, such as `goto`, `scanf` or `fscanf`, `fixed`, `strcmp_c`, `stricmp`, etc.
16. Do NOT use `void main()` -- use `int main()` always.
17. Do NOT use `return 0;` as the last statement in `int main()`, because it is not required by ANSI Standard C++.
18. If you use output statements in your program for debugging purposes, be sure to delete them before submitting your work.
19. Do *not* use "global variables".

Advanced checklist, even if your compiler "forgives" doing otherwise:

1. `NULL` is ONLY a pointer value -- do NOT use it for integer or other numeric values, even if your compiler allows it.
2. Do NOT declare a static array with a variable size specification, such as `int n =...; int a[n];`, even if your compiler allows it.
3. NEVER #include a CPP file, even if the textbook includes examples of such.
4. For console input and text file input with `>>`, use `.ignore` as explained in the Burns Intro to Programming text, chapter 5, unless specifically directed otherwise.
5. NEVER use `cin >>` to read directly into a numeric variable, like `int` or `double` or `float`. Read as a string (C or C++) and convert to a number using `atoi` or `atof`. Include `cstdlib`!
6. Do NOT use the `inline` keyword. To make class member functions inline, write their definitions inside the class definition.
7. Once the `const` keyword is introduced in this course, use it where applicable. Note that class "getter" functions are not really getter functions unless they have a trailing `const` keyword to flag them as such.
8. Make sure that all value-returning functions return a valid value under any logical circumstance.
9. Exit functions with `return` statements, and NOT `exit` -- do NOT use `exit` under any circumstances.
10. In every H and CPP file, ALWAYS #include libraries and H files for ALL functions, classes, and identifiers used in that file. Exception: use a class "forward declaration" instead of #including the class' H file when only pointers and reference variables appear.
11. Include the `cstring` library for char-array-based string ("C string") *functions*, and the `string` library for STL-based-strings ("C++ strings").
12. When using the random number generator seed function `srand`, call it ONCE ONLY in your program, and "sacrifice" the first value, like this: `srand(time(0)); rand();` at the top of main. Be sure to #include both `ctime` and `cstdlib`.
13. In COMSC 210, use `using namespace std;`. In 165 and 200, use `using std::` statements, properly placed under the #includes of the libraries to which they pertain.
14. `std::string` and `std::getline` belong to the `string` library. In 165 and 200, use `using std::` statements for these. Note that `cin.getline` is a different thing, and only requires `using std::cin;`.
15. Do NOT use `using std::` for anything in the C libraries. Use it for items from C++ libraries, only.
16. Test drivers MUST at least include ifndef testing, object copy testing with assignment after declaration, and const object testing with assignment upon declaration.
17. Test drivers must NOT include console or file input, unless directed otherwise.
18. Do NOT allow memory leaks, unless specifically instructed otherwise.