ENSF 694 – Summer 2025
Department of Electrical & Computer Engineering
University of Calgary

Lab Assignment 2

Written by: M. Moussavi, PhD, PEng.

This is NOT a group assignment, and you need to work individually.

Objective:

The objective of this lab is to understand the concepts and programming techniques such as:

1. Closer look at the C-strings
2. Recursion
3. Sorting Techniques
4. Complexity Analysis

Exercises in this lab are divided into two parts: Part I (closer look at the c-string, and understanding recursion, and sorting techniques, – writing C++ code), and Part II (complexity analysis, written answers),

Due Date: July 18, 2025, before 12 PM

Marking:

Part I (Algorithm Development)

Exercise A	3 marks
Exercise B	2 marks
Exercise C	not marked
Exercise D	12 marks (3 bonus mars for using GNU plot)
Exercise E	20 marks

Part II (Complexity Analysis)

Exercise A	7 marks
Exercise B	12 marks

Total: 56 marks

Part I – Algorithm Development

Exercise A (4 marks): Duplicating string library functions

Read This First:

A few facts about array of character and strings

Fact 1: C-string in an array of character, and the end of a string is marked by '\0', which is equivalent to integer 0. An array of character without null-terminator ('\0') is not considered a valid C-string and any operations or function calls that needs an argument of type 'valid C-string' may fail because of lack of '\0'. Here are a few examples:

```
char s1[6];
```

s1 is an array of character, and if it is use as an operand with `cout`:

```
cout << s1;
```

the result is undefined: For example, it may print garbage, or depending on different compilers, it may print nothing, or other undefined outcomes.

In the following example:

```
char s2[3] = "AB";
```
s2 is an array of character, and a valid C-string, because compiler copies string constant "AB" from string **constant area** on the **static segment** of the memory into the elements of s2, and puts a '\0' after the third element. As a result a `cout` statement works fine and prints the string: **AB**

In the following example:
```
char s3[3] = "XYZ";
```
s3 is an array of character that contains ABC but is NOT a valid C-string because compiler copies string constant "XYZ" from **string constant area** on the **static segment** into the elements of s3 but **cannot** put a '\0' after the string (because there is no more space left for '\0'. Therefore, the result of passing s3 to the `cout` is again undefined.

In the following example:
```
char s4[10] = "KLM";
```
s4 is an array of character that its first 3 characters are KLM, and the following elements are ALL '\0'. The reason is that, if even one element of the arrays is initialized the rest of them will be padded with zeros and they will not be garbage anymore.

Fact 2: Same as any other type of arrays, array of characters cannot be copied to each other. For example, the following code segment produces a compilation error.
```
char s4[10] = "KLM";
char s5[10];
s5 = s4;
```
The last line causes a compilation error. Therefore, if you really need to make s5 a copy of s4, you should do it element by element, using a loop:

```
for (int i = 0; i < 3; i++)
   s5[i] = s4[i];
s5[i]  =  '\0';    // added '\0' to make it a valid string
cout << s5;        // prints KLM
```

please notice that you need to add the '\0' manually to make s5 a valid C-string. Otherwise it will be only usable as an array of characters with KLM in the first three characters and the rest remains garbage.

It is also good to know that in the above code segment, I could also use Library function called strlen, instead using number 3 in the for loop:

```
for (int i = 0; i < strlen(s5); i++)
   s5[i] = s4[i];
s5[i]  =  '\0';
cout <<  s5;    // prints KLM
```

`strlen` returns the length of string, which in this example is 3. To be able to use C-String Library function such `strlen`, you must include the header file called `<cstring>`.

There is another way to make a C-string a copy of another C-string and that is by using a C-String Library function called `strcpy`. Here is an example:
```
char s4[10] = "KLM";
char s5[10];
strcpy(s5, s4);
cout << s5;    // prints KLM
```

Fact 3: We cannot use operators such as >, <, >=, <=, ==, or &&. || to have a logical operation on C-strings. For example, to compare two string we cannot use the following statement:
```
char s4[10] = "KLM";
char s5[10] = "ABC";
if(s4 >= s5) {
    // do something
}
```

This code segment gives logical error. When you are comparing the name of the two arrays s4 and s5, you are in fact comparing the addresses of the first elements of the arrays (remember: the name of the array is the address of the first

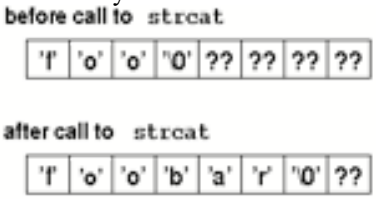
element). To be able to compare C-Strings there is a Library function called strcmp, the function returns a positive number, if its first argument is lexicologically greater than its second argument. It returns a negative number, if the second argument is greater than its first argument. Otherwise, if they are identical, it returns zero.

Fact 4: Unlike some other languages, the C-string doesn't do any boundary checking and if you try to write or read spaces before the first element, or after the last element, the compiler will not give you any compilation error. It is your responsibility to be aware of this fact and avoid any illegal operation.

Fact 5: Unlike some other languages, the C-string doesn't allow you to append strings using operators + or +=. You have to use the library function called `strcat`. Here is a very brief program that shows how `strcat`: works

```
int main(void){
  char s[8];
  strcpy(s, "foo");
  strcat(s, "bar");
  return 0;
}
```

This is what the array s would look like before and after the call to strcat:



Doing the concatenation involves three steps:

- Finding the '\0' character at the end of the destination string (the end of "foo" in the example).
- Copying all the characters from the source string ("bar" in the example).
- Adding a '\0' character at the end of the modified destination string.

A similar library function called `strncat` with the following function prototype:

```
char* strncat(char* dest, const char* source, int n);
```

Appends the first `n` characters of string source to string `dest`, and returns a char* to dest. If the length of the C- string in source is less than *n*, only the content up to the terminating null character, '\0', is copied.

Here are couple of examples:

```
char str1[20] = "abcd";
char str2[] = "xyz";
char str3[] = "MT";

strncat(str1, str2, 2);          /* appends xy the end of str1 */
cout << str1 << endl;            /* Prints: abcdxy */
strncat(str1, str3, 10);         /* appends MT to the end of str1 */
cout << str1 << endl;            /* Prints: abcdxyMT*/
```

There are more functions that can be used to manipulate C-strings.
Also, there are a few functions that you can be used to a single character. For example, a function such is islower returns true if its argument is a lower-case character:

```
  char s4[10] = "KLM";
  if(islower(s4[0])) {  // this statement returns true if first element is a
  lower case
      // do something
  }
```

In this exercise you are going to write your own version of some of the above-mentioned C-String library functions. In a practical programming project, writing a function that does the same job as a function in the library would be a serious waste of time. On the other hand, it can be a very helpful exercise for a student learning the fundamentals of working with C-strings.

What to do:

From D2L download the file `lab2exe_A.cpp`. Now study the set of slides called "Function Documentation", to understand what the requirements of the given functions are, also to follow the same principle to write function interface comment for each function that you will write in this exercise or future exercises.

Then, Study the file and write down your prediction for the program output. Compile the program and run it to check your prediction. If your prediction was wrong, try to understand why.

Make another copy of `lab1exe_A.cpp`; call the copy `my_lab1exe_A.cpp`. In `my_lab1exe_A.cpp`, add a function definition for `my_strlen` that calculates the length of a c-string. Then, replace all of the calls to `strlen` in the function main with calls to `my_strlen`, and then make sure your modified program produces the same output as the original.

Once `my_strlen` is working, add a function definition for `my_strncat`. *Don't make any function calls within your definition of* `my_strncat`. Fill in the missing parts of the function interface comment for `my_strncat`. Replace all of the calls to `strncat` in main with calls to `my_strncat`.

Submit the completed source file and your program output(s) that shows your program work, as part of your lab report on the.

Exercise B: Understanding Recursion

Read This First:

Mark Allen Weis the author of *Efficient C Programming*, has listed four fundamental rules of recursion:

1. *Base cases.* There must always be at least one case where the function can do its job without making a recursive call.
2. *Always make progress.* When a recursive call is made, it must make progress toward a base case.
3. *Believe that recursive calls work.* You should be able to convince yourself that a recursive function is correct without tracing execution all the way down to the base case.
4. *Some forms of recursion are terribly wasteful.* Badly designed recursive algorithms can do an enormous amount of redundant work.

Rules 1 and 2 must both be satisfied in order to ensure that infinite recursion does not occur.
Rule 3 is important. If you are trying to design a recursive function to solve a problem, you should think about how solving the problem involves solving a simpler problem of the same kind; you should *not* have to visualize all the activation records that will be required when the function runs.
A recursive function for computing a sequence of numbers known as Fibonacci numbers is the classic example for Rule 4.

What to Do:

Download file `lab2exe_B.cpp` from D2L. Compile and run the program to understand how simple function `sum_of_array` works. Now your job is to replace the given implementation of function `sum_of_array` with a recursive function (there may not be any loops in your function definition).

What to Submit:

Submit your source code as part of your lab report.

Exercise C: A slightly harder example

Download file `lab2exe_C.cpp` and write a function definition for the following function interface:

```
int strictly_increasing(const int *a, int n);
// REQUIRES:n > 0, and elements a[0] ... a[n-1] exist.
// PROMISES:
//   If n == 1, return value is 1.
//   If n > 1, return value is 1 if
//      a[0] < a[1] < ... < a[n-1]
```

```
//   and 0 otherwise.
```

Your solution must rely on recursion - there may not be any loops in your function definition.

There is nothing to submit for this exercise. However, please be aware that the exercises that are not marked are as important aa other exercises and similar problems may appear in the midterm or final exam.

Exercise D – A Recursive Method for Fibonacci Sequence (12 marks)

Fibonacci sequence is defined as: `0, 1, 1, 2, 3, 5, 8, 13, 21, 34 55, …`
Which can be formulated as below:

$F_0 = 0$
$F_1 = 1$
$F_n = F_{n-1} + F_{n-2}, \quad n > 2$

You may use the above formulae to calculate a Fibonacci sequence term, or you may define the following system of linear equations:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

Which can be expressed as:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

To implement this algorithm, you should write two functions as follows:

- Function called MutiplyMatrics that calculates the product of two N by N Matrices.
- Function called PowerMatrics that uses the following recursive definition to calculate matrix A to power n:

$$\text{If n is even: } A^n = A^{n/2} \cdot A^{n/2}$$
$$\text{Else: } A^n = A^{n/2} \cdot A^{n/2} \cdot A$$

What to Do:
Download the file `fibonacci.cpp` from D2L. If you study this file, you will see that definition of a few functions are incomplete and you must complete them. The program is supposed to use the clock function to approximate the process time for each method of calculation, and at the end uses Gnuplot to plot the process time for each of the two recursive and iterative function to show how the two methods have different rate of growth in terms of computation time. If for some reason you prefer not to use Gnuplot, you should use Excel to create your plot and submit it. 3 bonus mark will be considered for those using Gnuplot.

What is GNU plot?

GNUplot is a portable command-line driven graphing utility for various operating systems. Please, visit the official GNUplot website, the download page, and follow the instructions depending on your operating system to install it:

1. For Cygwin users: you need to run the setup executable `(setup-x86_64.exe for 64-bit or setup-x86.exe for 32-bit)`. Then, at the select packages screen, use the search bar to find gnuplot and follow the instruction to install it on your computer.
2. For Mac users: install Homebrew (if not already installed). Open terminal and use the following command to install gnuplot: `brew install gnuplot`
3. For Windows: Download the appropriate installer (a file ending with .exe) , run the installer and follow the instructions.

What to Submit:

The completed version of the Fibonacci.cpp, the program output, and the screenshots of two plots created by the Gnuplot or Excel.

Exercise E – Using Different Sorting Techniques

Read This First – How to Use C++ Library Function to Measure Program Execution Time

To measure the execution time of a process/function a possible option is to call function:
`std::chrono::high_resolution_clock::now()`, twice, once before and once after calling the function.
For more details, please see the description and example given at:
https://en.cppreference.com/w/cpp/chrono/high_resolution_clock/now

Then, the elapsed time is calculated by the differences of two times.

What to Do:
Download files `compare_sorts.h` and `compare_sorts.cpp`, In this exercise you are going to complete the missing functions in a program, which will produce a sorted list of unique words from a raw input text file. Here are more details about what the program is supposed to do:

1. Read in a word at a time from the input file, stripping out punctuation and white space (but allow hyphenated words), and convert all words to lower case. You can use C++ c-string and character library functions, as needed.
2. Store unique (i.e. non-duplicate) words in an array of strings. Assume that the maximum word size is 20 characters, and that no more than 10,000 unique words will have to be stored.
3. Sort the array into ascending alphabetical order, by calling three different sort methods (quick sort, shell sort, and bubble sort). Do not sort the string array directly, but sort the by an array of indices that refers to a word in an element of a string array.
4. For each sort method, write the sorted unique words into an output file, one word per line.
5. Calculate the time used to sort the words for each sorting method and report it on the screen.

Running your program:

Create your own input text file to test your code. An input file will be made available to the class a couple days before the due date. Use this file to create your output file and the timing reports.
All three-sort options should produce an identical output file. Depending on which operating system, you are using, you might be able to use a commend to compare the output file to make sure they are identical. For example, on Unix and Unix Like systems, you might be able to use the `diff` command.

What to Submit:
1. Your source code and the program output as part of your lab report.
2. Your actual source files, input and output file
3. The screenshot of your program output that shows 3 timing reports, like:

```
Sorting with Quick Sort completed in 0.000023 seconds.
Sorting with Shell Sort completed in 0.000035 seconds.
Sorting with Bubble Sort completed in 0.000121 seconds
```

Part II – Complexity Analysis

Exercise A:
Consider the following mathematical expression as function and order them based on their growth rate:

$N,\ \sqrt{N},\ N\log N,\ 2/N,\ 2^N,\ 2^{N/2},\ 37$

For each expression very briefly explain why is placed in your given order.

Exercise B:

For each of the following six program fragments indicate the Big O. Briefly explain why you suggest such an order.

```
(1)
sum = 0;
for( i = 0; i < n; ++i )
   ++sum;

(2)
sum = 0;
for( i = 0; i < n; ++i )
   for( j = 0; j < n; ++j )
     ++sum;

(3)
sum = 0;
for( i = 0; i < n; ++i )
   for( j = 0; j < n * n; ++j )
      ++sum;

(4)
sum = 0;
for( i = 0; i < n; ++i )
   for( j = 0; j < i; ++j )
      ++sum;

(5)
sum = 0;
for( i = 0; i < n; ++i )
   for( j = 0; j < i; ++j )
       for( k = 0; k < j; ++k )
          ++sum;

(6)
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < n; ++j )
           for( k = 0; k < n; ++k )
              ++sum;
```