ENSF 337: Programming Fundamentals for Software and Computer
Lab-3 - Week of September 27, 2021

Department of Electrical & Computer Engineering
Schulich School of Engineering
University of Calgary
Written and updated by: M. Moussavi, PhD, PEng

Acknowledgment: This lab contains some exercises written by Dr. S. Norman for previous versions of this course

Instructors:

- Dr. Mahmood Moussavi (B01, B02)
- Dr. Maan Khedr (B03, B04)

Objectives:

This lab consists of several exercises, mostly designed helping you to understand using and manipulating arrays and C-strings.

Due Dates:

B01	**Thursday Oct 7, 2:00 PM**
B02	**Thursday Oct 7, 2:00 PM**
B03	**Thursday Oct 7, 2:00 PM**
B04	**Tuesday Oct 5, 11:00 AM**

Marking Scheme:

Some exercises in this lab and future labs will not be marked. Please do not skip them, because these exercises are as important as the others in learning the course material.

Exercise	Marks
A	10
B	4
C	8
D	6
E	8
F	6

Total: 42 marks

Exercise A – Built in Arrays in C

Read This First - A Few Facts About Built in Arrays in C:

A built-in array in C is a data structure that can store a fixed size of sequential collection of elements of the same type. It is part of the language and doesn't need to include or import any library or header file. Here is a quick overview of a few facts about arrays in C:

Fact 1: When you declare an array with n elements of type T, as a local variable, a chunk of memory equal to the size of T multiplied by n will be allocated. In the following examples the size of x is 80 bytes, which is 8 (size of double) multiplied by10 (number of elements):

```
double x [10];                          /* size of x is: 10 * 8 = 80 bytes */
double y[] = {2.3, 3.0, 4.0};           /* size of y is 24 bytes */
```

C also provides an operator called `sizeof` that can be used to find the size of a data object in bytes. It can be applied either to a type or an expression. Here are examples of using `sizeof` operator:

```
int n = (int) sizeof(x);           /* n == 80 */
int m = (int) sizeof(double) * 10; /* m == 80 */
```

The value produced by `sizeof` operator is of type `size_t`, which is some sort of integer type; exactly which type it is depends on the particular implementation of C you are using. In this code segment we have used the type cast operator `(int)` to convert `size_t` type to the exact type of `int` on the left-hand side of the assignment operator.

The syntax `sizeof(`*`something`*`)` looks like a function call, but it isn't. When the compiler sees the expression `sizeof(x)`, it simply replaces the expression with the size of x in bytes.

Fact 2: Pointer and arrays are closely intertwined in C. Most of the time, when we use the name of an array in an expression, that name is automatically treated like a pointer to the first element of the array:

```
int ia[] = { 4, 6, 9};
int *ip = ia;
```

Here is an exception to this fact, when passing the name of array ia to the sizeof **operator** it is not treated as a pointer. It is treated just like an array – the value of y in the following example will be 12.

```
int y = (int) sizeof(ia);
```

Similarly, for proper type-match when the name of an array is passed to a function, the corresponding argument of the function has to be a pointer. For example a call to a function such as:

```
func(ia, 3);
```

Means the prototype of the function func should have two argument as follows::

```
void func(int*, int);
```

Fact 3: Arrays cannot be simply copied by using a single assignment statement that copies source array into an entire destination array. The following example produces a compilation error:

```
double x[3] = {7.5, 43.2, 0.3};
double y[3] ;
y = x;        /*ERROR */
```

Fact 4: Arrays in C cannot be resized. Therefore, they are often declared with a ``worst-case'' size. In the following example we assumed the maximum number of data at some point may or may not reach to 100, but at this point we are using only the first four elements of the array data:

```
double data [100] = {120.40, 200.00, 34.56, 99.88} ;
```

Fact 5: When we pass a numeric array to a function we should also pass an integer argument to the function indicating the actual number of elements to be used.

```
double x[10] = {2.50, 3.20, 33.0}; /* Note only first 3 elements of x are used */
double y[] = {5.00, 2.00};

my_function(x, 3);             /* my_function should use the first 3 elements */
my_fucntion(y, 2);             /* my_function should use entire array, 2 elements */
```

C-strings are exceptions: You don't have to worry about this exception in this lab -- we will discuss it during the lectures.

Fact 6: An array notation (square brackets) as a formal argument of a function is in fact a pointer. For example:

```
int foo(int a[], int n);
```

is *exactly* the same as:

```
int foo(int *a, int n);
```

What to Do:

Download the file `lab3exe_A.c` from D2L. Read the comment at the top of the file, and then try to predict the output of the program. Compile and run the program to check that your prediction of the output was correct.
Note: Some compilers may give warnings about size of pointers, but you still should be able to run the program.

Then,

1. Draw memory diagrams for point one, two, three, and four.
2. Add labels to diagram at point two to indicate the size in bytes for each variable, array, and function argument.

What to Submit:

Submit a properly scanned copy of your AR diagrams for point one, point two (with labels), point three, and point four as part of your lab report.

Exercise B: AR Diagrams with Arrays

What to do:

Download the file `lab3exe_B.c` from D2L. Read the file carefully. Predict the program output; build and run an executable to check your prediction. Make memory diagrams for point one, which appears within the definition of `reverse`.

Submit your memory diagram as part of your lab report.

Exercise C: AR Diagrams with C-String

What to do:

Download the file `lab3exe_C.c` from D2L. Read the file carefully. Predict the program output; build and run an executable to check your prediction.

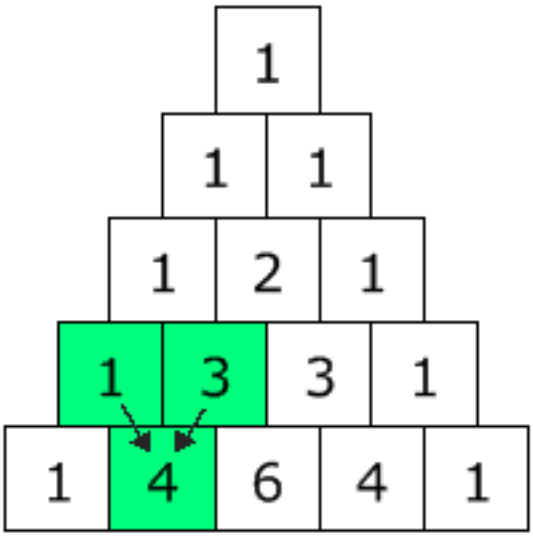
Make memory diagrams for point one and two. Please notice for point one you need to draw the diagram when program reaches this point for the first time.

Submit your memory diagrams as part of your lab report.

Exercise D - Problem Solving:

Read This First:

`Pascal's triangle` is a famous arrangement of numbers in mathematics. The first 5 rows of this triangle look as follows (row 0 to 5):



Let $\mathbf{P}_{i,j}$ be a number in row **i** of the triangle, and let the index **j** go from 0 up to **i**. Then:

$$\mathbf{P}_{i,\,j} = \begin{cases} \mathbf{1,\ if\ j = 0\ or\ j = i} \\ \mathbf{P}_{i-1,\,j-1} + \mathbf{P}_{i-1,\,j}\mathbf{,\ otherwise} \end{cases}$$

For example:

$P_{4,1} = P_{3,0} + P_{3,1} = 1 + 3 = 4$

What to Do:

Download `lab3exe_D.c` from D2L. Your job is to complete the definition of a missing function called `pascal_triangle`. This function is supposed to write the first N rows of Pascal's triangle on the screen, for values of N with N > 0 and N <= 20. The number of rows will be received from user with the given main function. Function `pascal_triangle` that receives the number of rows from main, as its argument, should print the triangle in the following format (figure shows Pascal's triangle for 6 rows):

```
Row 0:    1
Row 1:    1    1
Row 2:    1    2    1
Row 3:    1    3    3    1
Row 4:    1    4    6    4    1
Row 5:    1    5   10   10    5    1
```

Hint: Always you need to keep track of the values in two rows (current row and previous row of the triangle). Therefore in function `pascal_triangle` you can declare two integer arrays (say current-array and previous-array), with maximum N elements, plus some pointers that point to these arrays. After calculation of values in each row the current-array should be used as previous array to calculate the value of current-array. Also notice that when j == 0 or

j == i the values in the table are 1.

As part of your lab report, submit the definition of your `pascal_triange` *function and the program output for the case that triangle has 9 rows.*

Exercise E: Writing functions that work with arrays

Read This First:

In ENSF 337 exam we use function interface comments as an important part of program documentation. In this section a brief explanation in this regard is given. For more details please read the Function Interface Comment posted under the "Help Documents for Lab Assignments" on the D2L.

Lets consider the following function interface comment:

```
int largest(const int *a, int n);
/* REQUIRES
 *   size_a > 0.
 *   Elements a[0], a[1], ..., a[n - 1] exist.
 * PROMISES
 *   Return the largest value of a[0], a[1], ... a[n - 1]. */
```

This function obviously requires that value of n not to be negative. However the next line:

```
      Elements a[0], a[1], ..., a[n - 1] exist.
```

says that array a is big enough to have elements with indices up to and including size_a-1. When certain requirements are mentioned under the required, it means that, this function is not responsible to test if the value of n is negative or if the array's memory space is adequate or not (no error checking for those requirements are needed). As we discussed during the lectures, C compilers cannot check if someone tries to do something with the elements of an array beyond the boundaries of an array. Nor there is any way that you can check such a condition.

What to do:

Download the file `lab3exe_E.c` from D2L. In this exercise the definition of two functions are missing and as result the program cannot do something useful. Your job in this exercise is to take the following steps to complete the definition of the missing functions:

Step 1. Read the file `lab3exe_E.c` carefully. Pay attention to the syntax of the function prototypes, function calls, and function definitions.

- Note that whenever any function uses values from an array but does not modify those values the corresponding argument type is `const int*` instead of simply `int*`.
- Note that when reading input, if there is an invalid entry, it ignores those characters and removes them from the input buffer.

Step 2. Compile it and run it. Find out how the program works.

Step 3. The function definition for `select_negatives` is incomplete. Read its interface comment and complete the definition of this function. Compile, run, and test it to make sure it works.

Step 4. The function definition for `substring` is also incomplete. Read its interface comment and complete the definition of this function. Compile, run, and test it to make sure it works.

Note: The purpose of this exercise is to learn how to manipulate c-strings. Therefore you are not allowed to use C/C++ library functions such as `strstr,` or `strncat`.

Submit your source code and the program outputs to demonstrate that your program works for different set of data.

Exercise F: More Practice with Strings

Read This First –I/O Redirection

A common method that users can send input to a C program is by entering the input on the keyboard, and pressing the 'enter/return key'. However the other powerful feature is to redirect the program input to a file. In other words, we can enter a text-file name on the command line after an input-redirection operator, <, and when the program is expecting to read some input from keyboard it will use the content of the given file as user's entry. Similarly, you can use the output-redirection operator, >, to send the program's standard output (the program output that normally is displayed on the screen) to a text file. Let's assume we have a text file called `input.txt` that contains the following data:

```
red
pink
yellow
apple
junk
pear
123
orange
```

Now, while you are in the same directory that file `input.txt` is located, use the Linux sort command (on the Cygwin command line), in the following format to produce a text file called `sorted_nput.txt`:

```
sort input.txt<input.txt>sorted_input.txt
```

Now if you open the file `sorted_input.txt`, the content of the file will be:

```
123
apple
junk
orange
pear
pink
red
yellow
```

We are going to use this feature in this exercise to read a text file into a C program and indicate if each sentence in the given input file is a palindrome phrase or not.

Read This Second:

A **palindrome** is a word, phrase or number which has the property of reading the same in either direction. The word "palindrome" comes from the Greek words *palin* ("back") and *dromos* ("racecourse"). Here are some examples:

```
"Madam I'm Adam."
"Radar"
123321
Can I attain a 'C'?
Can I attain a $$C?
```

As the example shows the adjustment of leading spaces, trailing spaces and spaces between letters is generally permitted. The removal of any non-alphanumeric characters (such as punctuation or random $ signs) is also allowed. In the above examples if we remove all of the non-alphanumeric characters and convert all letters to lower case, the strings will be spelled the same from both ends:

```
madamimadam
radar
123321
caniattainac
caniattainac
```

What to do

1. Copy files `palindrome.c`, and `palindrome.txt`, from D2L.
2. Read the files and understand what the program is doing.
3. Build and run the program.
4. If you are running the program in ICT 320, while you are in your working directory enter the following command (I assume your executable file is `a.exe`):

   ```
   ./a.exe < palindrome.txt
   ```

   If you are working on a Windows machins click Start Button, and type 'cmd' into the box: Search Programs and Files. Then you should see a window with a black background and allows you to enter commands. Again you have to make sure you are in the working directory for this exercise before running your program.

   The program should produce an output which its first three lines are:

   ```
   "Radar":  is not a palindrome.
   "Madam I'm Adam":  is not a palindrome.
   "Alfalfla":  is not a palindrome.
   …
   ```

   Certainly something is wrong in this program! None of the above strings are indicated as palindromes. If ignoring the lower/upper case and spaces, etc. the first two statements must been detected as palindrome.

   Note: Although not necessary in this exercise, but you could also redirect the program output into another file if you wish.

5. Take the following steps to fix this problem:

   a. Uncomment the lines commented out in the main function, confined between `#if 0` and `#endif`. It means you should change `#if 0` to `#if 1` (Note: we will discuss and learn more about how `#if 0` works in near future).
   b. Read the function interface comments in `palindrome.c` and write the definition of the the two functions: `is_palindrome` and `strip_out`.
   c. Compile the program and run it again. If your functions work, the following output should appear on the screen:

      ```
      "Radar":  is a palindrome.
      "Madam I'm Adam":  is a palindrome.
      "Alfalfa":  is not a palindrome.
      "He maps spam, eh?":  is a palindrome.
      "I did, did I?":  is a palindrome.
      "       I prefer pi.":  is a palindrome.
      "Ed is on no side":  is a palindrome.
      "Am I loco, Lima?":  is a palindrome.
      "         Bar crab.":  is a palindrome.
      "A war at Tarawa.":  is a palindrome.
      "Ah, Satan sees Natasha":  is a palindrome.
      "     Borrow or rob?":  is a palindrome.
      "233332":  is a palindrome.
      "324556":  is not a palindrome.
      "Hello world!!":  is not a palindrome.
      "     Avon sees nova  ":  is a palindrome.
      "Can I attain a 'C'?":  is a palindrome.
      "Sept 29, 2005.":  is not a palindrome.
      "Delia failed.":  is a palindrome.
      ```

```
      "Draw nine men $$  inward":  is a palindrome.
```

Note: In this exercise you can use C library functions such as:

```
strlen(s);                  /* returns length of string s */
islower(ch);                /* returns 1 if character ch is a lowercase letter, 0
                               otherwise */
isupper(ch);                /* returns 1 if character ch is an uppercase letter,
                               0 otherwise */
ch = tolower(ch);           /* converts ch to lower case  */
isalnum(ch);                /* Returns 1 if ch is an alphanumeric character,
                               0 otherwise */
```

Submit your source code and the program outputs to demonstrate that your program works.