ENSF 614 – Advanced System Analysis and Software Design
Lab-2 – Winter 2023

Instructor: Anja Slama (anja.slama@ucalgary.ca)
Department of Electrical & Computer Engineering
University of Calgary

Notes: In this lab, you are allowed to work with a partner (ONLY groups of two. groups of three or more are NOT allowed). If you decide to work with a partner, please submit only one lab report with both names on the report.

Due Date: 11:59 PM on Friday January 27. All your work should be in a single PDF file that is easy for your TA to read and mark.

Objectives:

This lab consists of several exercises, mostly designed to help you to understand using arrays and built-in strings, pointer arithmetic, and struct data types.

Marking Scheme
You should not submit anything for the exercises that no marks are assigned to them.

Exercise	Marks
A	8 marks
B	8 marks
C	No marks
D	No marks
E	6 marks

Total marks: 22 marks

Exercise A – Built in Arrays

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

A built-in array is a data structure that can store a fixed size of a 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:

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 by 10 (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 the compiler 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);
```

This means the prototype of the function func should have two arguments as follows::

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

Fact 3: Arrays cannot be simply copied by using a single assignment statement that copies the 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 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);` And both are exactly the same as:

```
int foo(int a[100], int n); // compiler ignore the number 100 between []
```

What to Do:

Download the file `lab2exe_A.cpp` from D2L. Read the comment at the top of the file, and then try to predict the program's output. Compile and run the program to check that your prediction of the output was correct. **Note:** Some compilers may give warnings about the size of pointers, but you still should be able to run the program.

Then,

1. Draw memory diagrams for points one, two, three, and four.
2. Add labels to the 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 (8 marks): Duplicating string library functions

Read This First:

A few facts about arrays of characters and strings

Fact 1: C-string in an array of characters, and the end of a string is marked by '\0', which is equivalent to integer 0. An array of character without a 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 characters, and if it is used 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 characters, and a valid C-string, because the compiler copies string constant "AB" from string **constant area** on the **static segment** of the memory into the elements of s2 and put 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 characters that contains ABC but is NOT a valid C-string because the 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 characters 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 array, an 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 used 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 a Library function called `strlen` instead of using the 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 the string, which in this example is 3. To use C-String Library function such as `strlen`, you must include the header file called `<cstring>`.

Another way to make a C-string a copy of another C-string. 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 strings we cannot use the following statement:

```
char s4[10] = "KLM";
char s5[10] = "ABC";
if(s4 >= s5) {
// do something
}
```

This code segment gives logical errors. For example, when you are comparing the name of the two arrays, s4 and s5, you are comparing the addresses of the first elements of the arrays (remember: the name of the array is the address of the first element). 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. You are responsible for being aware of this fact and avoiding any illegal operation.

Fact 5: Unlike some other languages, the C-string doesn't allow you to append strings using the 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`:

before call to `strcat`

'f'	'o'	'o'	'\0'	??	??	??	??

after call to `strcat`

'f'	'o'	'o'	'b'	'a'	'r'	'\0'	??

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 the source is less than *n*, only the content up to the terminating null character, '\0', is copied.

Here are a 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*/
```

More functions 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 the first element is a lower case
// do something
}
```

In this exercise, you will 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_B.c`. Study the file and write down your prediction for the program output. Then compile the program and run it to check your prediction. If your prediction was wrong, try to understand why.

Make another copy of `lab2exe_B.c`; call the copy `my_lab2exe_B.c`. In `my_lab2exe_B.c`, 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 make sure your modified program produces the same output as the original.

Once `my_strlen` is working, add a function definition for `my_strncat` to `my_lab2exe_B.c`. *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`.

Your other task in this exercise is to add a function prototype, function interface comment, and the function definition for a function called `my_strcmp`. This function should work like `strcmp` library function.
If you don't already know the details of how `strcmp` works, here is a closer look at this library function:

The function prototype of this function is:

```
int strcmp(const char* str1, const char* str2);
```

`strcmp` compares `str1` and `str2`, and returns 0 if the two strings are identical. Otherwise, returns a positive number if `str1` is greater than `str2`, and a negative number if `str2` is greater than `str1`.

Method one: In some platforms, the return value of `strcmp` is:

- `1, if str1 is "ADC" and str2 is "ABC".`
- `-1, if the str1 is "ABC" and str2 is "ADC".`
- `0, if the str1 is "ABC" and str2 is also "ABC".`

Method two: Some other compilers may operate differently. They may return 0 when `str1` and `str2` are identical. Otherwise, they may return the ASCII value differences of the first two characters that are different. For examples:

- `If str1 is "ADC" and str2 is "ABC", it returns 2.`
- `If the str1 is "ABC" and str2 is "ADC", it returns -2.`
- `If the str1 is "ABC" and str2 is also "ABC", it returns 0.`

If you try the following code on one of the recent Mac computers:

```
char str1[10] = "ABCDE"; char
str2[10] = "ABCD";
int y = strcmp (str1, str2);
```

The value of `y` will be 69, because `str1[4] == 'E'` which its ASCII value is 69 and `str2[4] == 0`.

For this exercise, your function `my_strcmp` should use method two.

Submit the completed source file and your program output(s) showing your program work as part of your lab report.

Exercise C: Pointer arithmetic

What to do:

Download the file `lab2exe_C.cpp` from D2L. Read the program and draw a memory diagram for the second time the program gets to point one, which occurs during the second call `main` makes to function `what`. If you have difficulty tracing the program, you may want to add some calls to `cout` statements in `what`, then run the program to collect extra information.

This exercise will not be marked, and you shouldn't submit anything.

Exercise D: Drawing pictures of `struct` variables

Read This First:

A structure type is a type that specifies the format of a record with one or more members, where each member has a specified name and type. These members are stored in memory in the order that they are declared in the definition of the structure, and the address of the first member is identical to the address of the structure object itself. For example, if we consider the following definition for structure `Course`:

```
struct Course

{

char code[5];

int number;

char year[4];

};
```

And the following declaration of an instance of `struct course`:

```
Course my_course = {“ENCM”, 339, “2nd”};
```

The address of member `my_course.code` is identical to the address of `my_course`, and the address of member `my_course.number` is greater than the address of the previous member, `my_course.code`. However, the address of the member `my_course.number` will not necessarily be the address of the following byte right after the end of memory space allocated for the member `my_course.code`. It means there might be gaps or unused bytes between the members. The compiler may align the members of a structure for certain kinds of the addresses, such as 32-bit boundaries, to ensure fast access to the members. As a result, the size of an instance of the structure, such as a course, is not necessarily equal to the sum of the size of the members; it might be greater. In summary: the size in bytes of a `struct` is always greater than or equal to the sum of the sizes of its member variables.

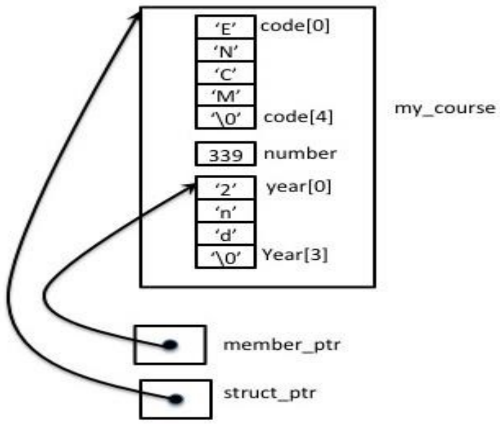
Read This Second – Structures and Pointers

In principle, a pointer to a structure type is not much different from other types of pointers. They are supposed to hold the address of a `struct` object and are of the same size as other pointers. It is important to realize that no matter how big and complicated a `struct` type is, pointers to that `struct` type are small and simple. Remember that the address of a variable is the lowest address of all the bytes used to store it. So, the address of a `struct` variable is the address of the first byte of the region of memory used to store all the members.

Please notice, when drawing AR diagrams, make sure to be clear whether the arrowhead points to the entire `struct` instance or a member of the structure. Please see the following example:

```
Course* struct_ptr = & my_course;

char* member_ptr = mty_course.year;
```



What To Do

The main point of this exercise is to help you visualize how `struct` variables and pointers to `struct` variables are organized on the computer memory.

Download the file `lab2exe_D.cpp` from D2L. Read the program carefully and try to predict the output. Run the program. Ask a lab instructor or teaching assistant for help if you are surprised by any of the output. Draw stack diagrams for point one and point two.

This exercise will not be marked, and you shouldn't submit anything.

Exercise E (6 marks): Writing Functions with struct Type

Read This First

In general, most real-world C++ programs are divided into two different parts: The implementation part normally consists of one or more source code with the extension of `.cpp`, and the interface part may include one or more header files with the extension `.h`.

The .h files may contain a declaration/prototype of one or more functions and some other general and global definitions or declarations, such as:

- Declaration of global constants
- Definition of user-defined types such as `struct` data types
- pre-processor: `#include, #define`, etc.

The header files the programmer will create must be in the following format, confined between pre-processors typed in red.

```
#ifdef XYZ // you can replace term XYZ with any other word that you like
#define XYZ // whatever term is used in previous line must be exactly used here
...    // your items such as global constants, function prototypes, goes here
...    // etc.
#endif
```

The .cpp files normally should contain the definition/implementation of the functions. They must include the `.h` file that contains the function prototypes and other items as indicated above, using a pre-processor directive. When we include the system header files like `<iostream>`, we use angle-bracket `<    >`. However, when we include our own header files or other programmers' header files, we must use double quotation marks:
`#include "your_file_name.h"`

To better understand the format of .h files, here is a partial example that shows two files `point.cpp`, and `point.h`

Header file	Implementation file

`// point.h`
`#ifndef MY_POINT`
`#define MY_POINT`
 	
`// definition of structure Point`
`struct Point {`
`  double x;`
`  double y; // Point's y-coordinate`
`  char label;`
`};`
 	
`// function prototypes`
`void foo(Point *p, int x, int y);`
`void bar(int *a, int n);`
 	
`// declaration of a global constant const`
`double PI = 3.145;`
 	
`#endif`
	`// point.cpp`
	`#include "point.h"`
	
	`int main(void)`
	`{`
	`   Point centre;`
	`   int a, b;`
	`   foo(centre, a, b);`
	`   ...`
	`   bar(a, b);`
	`   return 0;`
	`}`
	`void foo(Point *p, int x, int y)`
	`{`
	`    // implementation of function foo`
	`}`
	`void bar(int *a, int n)`
	`{`
	`     // implementation of function bar`
	`}`

Read This Second

Complex numbers have the form `a + j b`, where `a` and `b` are real numbers and `j` is a square root of -1. The quantity `i` is called the real part, and the quantity `b` is called the imaginary part. Here is a summary of the rules for complex arithmetic:

$$(a + j\,b) + (c + j\,d) = (a + c) + j\,(b + d)$$

$$(a + j\,b) - (c + j\,d) = (a - c) + j\,(b - d)$$

$$(a + j\,b)\,(c + j\,d) = (ac - bd) + j\,(ad + bc)$$

The complex number module you will work on in this exercise has a terrible, awkward design. The point of this bad design is to force you to think about some of the different ways functions can work with `struct` variables--using value arguments, pointer arguments, and return statements. Don't use this as a design model!

What To Do

Download files `lab2exe_E.cpp`, `lab2exe_E.h`, and `lab2exe_E_main.cpp` from D2L.

Step 1. Read through the three source files. Build an executable and run it. To compile this program, you should enter the following command.

```
g++ -Wall lab2exe_E.cpp lab2exe_E_main.cpp
```

Note: You only compile `.cpp` files. The header file `lab2exe_E.h` **should not** appear in the command.

Step 2. In the file `lab2exe_E.cpp` the definition of `cplx_add` is given. You should add the definitions for `cplx_subtract`, and `cplx_multiply`.

Do NOT change the prototypes for any of these functions; one of the points of this exercise is to get used to the different methods available for working with `structs` and functions.

Step 3. Add code to main to call the functions you wrote in Step 2 and print out the values of `w-z, and w*z`. Run your program using input values of `w = 1.5 + j 0.75`, and `z = -2.5 - j 0.5`. Check the results by hand or with a calculator.

What To Submit:

Submit only the definition of three functions: cplx_add, cplx_subract, and cplx_multiply, as part of your lab report on the D2L.