ENSF 337: Programming Fundamentals
Lab 2 – Thursday September 20, 2018
Department of Electrical & Computer Engineering
University of Calgary
M. Moussavi

Acknowledgement: This document includes some materials written by Steve Norman in previous years.

Objectives:
This lab consists of several exercises, mostly designed helping you to:
- Practice drawing AR diagrams
- Understand simple application of pointers in C

Please heed this advice:
- You are strongly advised to study lab documents before coming to your scheduled lab sessions and at least complete the first few exercises at home.

- 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.

- In courses like ENSF 337 some students skip directly to the exercises that involve writing code, postponing the diagram drawing until later. That's a bad idea for two reasons:

    - Drawing diagrams is an important part of learning how to visualize memory used in C and C++ programs. If you do diagram-drawing exercises at the last minute, you won't learn the material very well.

    - If you do the diagrams first, you may find it easier to understand the code-writing exercises, so you may be able to finish them more quickly.

Due Date:
In every scheduled lab session in ENSF 337 you will find two types of lab exercises:
- In-lab exercises that must be always handed in on paper **by the end of your scheduled lab period**, in the hand-in boxes. The hand-in boxes are on the second floor of the ICT building, in the hallway on the west side of the building.
- Post-lab exercises normally must be submitted electronically using the D2L Dropbox feature. All of your work should be in a single PDF file that is easy for your TA to read and mark.
- **Due dates and times for post-lab exercises are before: Thursday September 27 2:00 PM**

Note: 20% marks will be deducted from assignments handed in up to 24 hours after each due date. It means if your mark is X out of Y, you will only gain 0.8 times X. There will be no credit for assignments turned in later than 24 hours after the due dates; they will be returned unmarked.

Marking Scheme:

You shouldn't submit anything for the exercises that are not marked.

Exercise	Type	Marks
A	in-lab	8
B	in-lab	5
C	in-lab	4
D	post-lab	3
E	post-lab	9
F	post-lab	10
G	post-lab	10

Exercise A – Introduction to Pointers

This is an in-lab exercise

Read This First

This is an important exercise in ENSF 337. If you don't become comfortable with pointers, you will not be able to work with C. Spend as much time on this exercise as is necessary to understand exactly what is happening at every step in the given program.

What to do

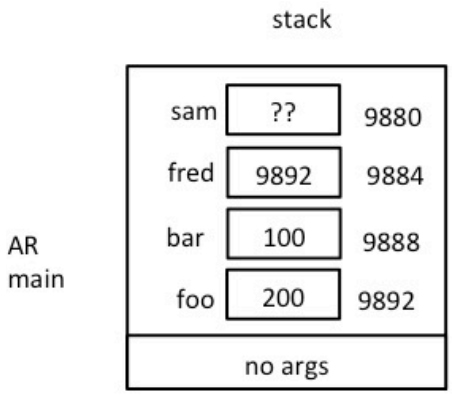
Download the file `lab2exe_A.c`. With pencil and paper, trace through the execution of the program to determine what the output will be.

Now assume the following addresses for variables:

```
sam   9880
fred  9884
bar   9888
foo   9892
```

Draw a set of AR diagrams for points two through five and use the given address numbers as values of the pointers (don't use arrow notation in this exercise).

To understand how to draw these diagrams the solution for point one is given in the following figure:



After you completed the exercise, compile lab2exe_A.c and check your predicted output against the actual program output.

What to Submit:

Submit your diagrams as part of your in-lab report.

Exercise B: Pointers as function arguments

This is an in-lab exercise

What to do – Part I

First copy the file `lab2exe_B1.c` from D2L. Carefully make diagrams for point one in this program using "**Arrow Notation**" as we discussed during lectures (you don't need to use made-up addresses). Then compare your solution with the posted solution on D2L.

There is nothing to submit for this part

What to do – Part II

Now download the file `lab2exe_B2.c` from D2L and draw AR diagram for point one in this file.

What to Submit:

Submit the AR diagram for part II as part of your in-lab report.

Exercise C: Using pointers to get a function to change variables

This is a in-lab exercise

What to do

Make a copy of the file `lab2exe_C.c` from D2L. If you try to compile and run this program it will give you some warning because the definition of function `time_convert` is missing.

Write the function definition for `time_convert` and add code to the main function to call `time_convert` before it prints answers.

Submit `lab2exe_C.c` and your program's input and output as part of your in-lab report.

About the function interface comment

The line minutes_ptr and seconds_ptr point to variables, appears in the REQUIRES section for the time_convert function. You might think that's useless information, but it's not. It is quite possible for a pointer argument to contain a garbage address, or a null pointer value, and in either of these cases the program would almost certainly crash. (You'll learn about null pointers later in the course.)

Exercise D: More on `scanf`

This is post-lab exercise. Only part II will be marked

Read This Firs:

`scanf` returns an integer number which indicates the number of input items that `scanf` has successful read from the keyboard. For example in the following code segment `scanf` is supposed to read three integer numbers and put them into the variables a, b, and c.

```
int a, b, c, nscan;
printf("Please Enter three integer values: ");
nscan = scanf("%d%d%d", &a, &b, &c);

if (nscan != 3) {
   printf("Error: invalid input(s). I quit.\n\n");
   exit(1);
}
```

To detect input error after calling `scanf`, a program must check the *return value* from `scanf`. If user enters invalid input, the return value of the `scanf`, `nscanf`, will have a value of zero, one, or two depending on how many of the inputs were properly entered. In this example, if `nscanf` is not equal to 3 the program will terminate using the exit() library function.

In general, scanf returns the number of items that it reads successfully, or returns EOF. EOF is a negative constant defined in <`stdio.h`>.The value EOF means that `scanf` reached the end of an input file or that some other error occurred.

An important concept to understand about `scanf` (and many other C input functions) is that a C program sees input as a stream of characters, not as a sequence of lines of text. `scanf` will consume just enough characters to do its

job, or to find out that it can't do its job. Remaining characters on the input stream are not consumed--instead they remain in the stream, waiting for the next input operation.

Read This Second:

Before proceeding to the rest of this exercise, note that if a program goes into an infinite loop, or `hangs' in some other way, you can kill it by typing `Ctrl-C` (holding down the Ctrl key and pressing C).
(By the way, `Ctrl-Z` `stops' a running program. In Unix and Unix-like operating systems such as Linux, stopping a program means suspending the program so that it can be resumed later. If you are trying to kill a program, you should use `Ctrl-C`, not `Ctrl-Z`.). Also note that on these systems, typing `Ctrl-D` at the beginning of a line of input indicates end-of-file on the keyboard input stream.

Note: typing `Ctrl-D` works only from command line. Meaning from within development tools such as Visual Studio, Xcode, or Eclipse doesn't work.

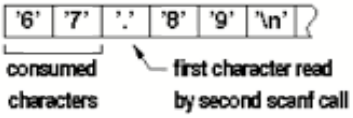
What To Do - Part I;

Copy the file `lab1exe_D1.c` from D2L. Read the program. Compile it, then run it several times, first with sensible input, then with various forms of bad input.

For example observe what happens when you enter letters in response to the prompt for an integer? What if you type two integers on the same line? What if you type 67.89? What if you type one integer followed by letters? (You don't have to write down the answers.)

An Explanation:

The following picture shows what happened when you typed in 67.89 in response to the prompt for an integer. The area of the memory that stores the stream of character inputs is called **'buffer'**.



The picture shows the stream of input characters after the first call to `scanf` was complete. Here is what the first call did:

- Read the '6', saw that it was a digit and could be used as part of an int.
- Read the '7', saw that it was a digit and could be used as part of an int.
- Read the '.', saw that it was not a digit and could be not be used as part of an int. The '.' was put back on the input stream so that it could be read by the next input operation.

This is an important exercise. Don't hesitate to ask instructor or a TA for more information. You will not be able to complete the next exercise if you do not fully understand this exercise.

What to Submit:

There is nothing to hand in for this exercise (part I).

What to Do – Part II:

Copy the file `lab1exe_D2.c` from D2L. Read the program. Compile it, then run it for 6 times, enter the inputs as listed in the following table, and complete the table for the values of n, i, and d.

Run #	Your inputs	What is the value of n	What is the value of i	What is the value d
1	`12 0.56`			
2	`5.12 9.56`			
3	`12 ab`			
4	`ab 12`			
5	`5ab 9.56`			
6	`13 67`			

What to Submit for part II:

This part will be marked. You should submit the above completed table.

Exercise E: Simple functions to do input error handling

This is a post-lab exercise

Read This First

The exit function is part of the Standard C Library. Its prototype is: `void exit(int status);`

You make the compiler read the prototype for exit by putting `#include <stdlib.h>` in your source code.

A call to exit terminates the execution of a program. Don't confuse the exit function with the break statement, which terminates the execution of a loop or a switch statement, or with the return statement, which terminates the execution of a function.

The argument to exit is used to provide information about whether the program succeeded or failed, and sometimes about the way in which the program failed. (This is the same information that can be communicated via the return value of main.) Knowing exactly how to use this feature of the programming environment is only necessary when you are developing a system in which certain programs control the execution of other programs. Nevertheless it is good style in the UNIX and MS-DOS environments to use exit(0) to indicate successful termination and exit(1) to indicate termination because of an error condition.

In this exercise you will write functions that try to read numbers using scanf. The functions will implement a very simple error-handling policy: if there is bad input, the program should print an error message and die. This policy, although crude, is much better than simply letting the program proceed with garbage values in some of its variables.

What to do

Make a copy of the file `lab1exe_E.c`. Observe that it contains two function prototypes, a main function, and two incomplete functions. Your job is to complete the function definitions for `read_int` and `read_double`. Your goal is not to develop a program that does useful work; rather, your goal is to build and test two functions that could be useful parts of other programs.

***Useful fact**:* The `scanf` conversion specification for the double type is `%lf`; `%f` usually will not work. This is one of many situations where `printf` and `scanf` use different conversion specifications for the same type.

If both your functions are properly working, the program should not accept any of the followings as an entry for an integer number, and after giving an error message should terminate the program (using library function exit):

- `23.44`
- `23abc`
- `23.`
- `23+44`

Your program should not also accept any of the following inputs as a double number. It should give an error message, and terminate the program (using library function exit):

- `23..44`
- `ab2.3`
- `23.4a`
- `23.5+`
- `abcd`

Hints:

- Develop incrementally. Get `read_int` working first. Once that problem is solved, move on to `read_double`.
- When the user enters `23abc` (i.e., an invalid integer) and `scanf` is supposed to read an integer, it consumes 23 and uses 23 as an integer (i.e., considers 23 as valid input and ignores abc). However, `abc` is still left in the buffer. Your program should be smart enough to realize that there is an invalid entry (in this case abc), using function `fgetc` that reads the next character from the buffer. Assume that the user's entry after the last digit of a number always ends with either `'\n'` or `EOF`. `EOF` is a predefined constant, used as `End Of File` indicator.

  .

  One major difference between `fgetc` and `scanf` is that `scanf` does not read white space (space, tab, return), while `fgetc` reads all character, including white space characters. Here is an example of using `fgetc`:

  ```
  int c;
  c = fgetc(stdin);
  ```

  Note: `stdin` is a predefined pointer that will be discussed later during the lectures. You don't need to worry about the details of this pointer in this lab.

What to Submit:

Submit your completed program and the program output that shows your program works.

Exercise F: Writing a Complete C Program with a Few Functions

This is a post-lab exercise

What to do

Write a program that asks a user for a distance in kilometres and vehicle's average speed, and its output is the travel time in hours and minutes. To give you a better idea, here is three samples of dialogues between user and program:

Sample run #1:

```
Please enter the travel distance in km: 50

Now enter the vehicle's average speed (km/hr): 100

You have travel 50.00 (km) with a speed of 100.00 in 0.00 hour(s) and 30.00 minute(s)
```

Sample run #2:

```
Please enter the travel distance in km: 100

Now enter the vehicle's average speed (km/hr): 50

You have travel 100.00 (km) with a speed of 50.00 in 2.00 hour(s) and 0.00 minute(s)
```

Sample run #3:

```
Please enter the travel distance in km: 125.5
```

```
Now enter the vehicle's average speed (km/hr): 65.5

You have travel 125.50 (km) with a speed of 65.50 in 1.00 hour(s) and 54.96 minute(s)
```

You should organize the program to have three functions as follows, plus a main function:

`get_user_input` that returns nothing and reads the user input, distance in km, and vehicles speed in km/hr.

`travel_time_hours_and_minutes` that returns nothing, receives the user inputs (distance and speed), and calculates the travel time in hours and minutes.

`display_info` that is void function (no return value), and its only responsibility is to display the distance that was traveled, followed by the speed of vehicle, and then the travel time in(hours and minutes)

More Hints: Here are the function prototypes to used in your program:

```
void get_user_input(double* distance,  double* speed);

void travel_time_hours_and_minutes(double distance, double speed, double *hours,
                                                                    double *minutes);

void display_info( double distance, double speed, double hour, double minutes);
```

In addition to above sample run input, you should test your program several time. One of the test cases must be 5.44 km average speed 76.5.

Submit your code and your program's input/output.

Exercise G – Built in Arrays in C

This is post-lab exercise.

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 `lab1exe_G.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 post-lab report.