Lab Exercises: LAB 4
(Pointers & Memory Management)

General guidance:

1. You will need to log on to your Linux environment for this lab.
2. Use a text editor of your preference to edit the code. For today's lab you will not need anything more complicated.
3. For quick on-line reference use (a book is preferable): *http://www.cppreference.com/*
4. Use the g++ compiler at the command line: For example `g++ ex1.cc -Wall -o ex1` will compile the source file ex1.cc and will generate the executable code in the program file ex1.
5. You can run a program from the directory where you compiled it by typing (for the above example): ./ex1
6. Use the `mkdir` command to create a directory structure like, e.g. *`your_home/CIP/labs/lab4`*. If unsure ask for help. Then `cd` to `lab4`.

> **NOTE:** It does not matter what file extension you use for your source files, .cc, .cpp – It is a matter of stylistic preference.

Compiling multi-file projects

In case where your program consists of more than a single .cpp file, you can still compile all the cpp files in a way similar to the example given below:

```
g++ -o main main.cpp building.cpp house.cpp
```

(for the example of exercise 6 from Lab1). This will compile and link the files all at once.
Note: We do not list the header files. Since they are included into the above files, they will get compiled. Also, the `-o main` tells it to generate an executable named main instead of the default `a.out`.

However, you can also compile the source files separately, and then link them together to the executable file. This will allow us to only regenerate the parts of the program that are affected when we change some source code.
When we just compile source code (without linking it together), it means that we take the .cpp files and generate intermediate object files (.o).
To just compile source code, use the -c flag with the compiler:

```
g++ -c main.cpp
g++ -c building.cpp
g++ -c house.cpp
```

This will generate the object files `main.o, building.o` and `house.o` respectively.
Then, we can link the object files together by using:

```
g++ -o main main.o building.o house.o
```

In this separate linking and compiling, if we subsequently change only e.g. the file `main.cpp`, then we only need to recompile this file and then link all the .o files together.
Note: Whether a .cpp file changes or a header file included (directly or indirectly) by that .cpp file changes, we have to regenerate its corresponding .o file.

Exercise 1.

Using the lecture notes from last week, make sure you try all the pointer examples and that you feel comfortable using the new operators: * and &.

Exercise 2.

Using parameter passing by reference, implement a simple function `void increment` that when called as `increment(n1, n2, inc)` will locally (i.e. in the function) add `inc` to both `n1` and `n2`, but will actually alter ("globally") the value of only `n2`.

Write a simple `main` function to call the `increment` function, and insert appropriate output commands to check the values of the variables `n1` and `n2` before, during and after the call of the `increment` function.

Exercise 3.

Using arrays with dynamic memory allocation, write a program that simply "memorises" a list of numbers at input (i.e. stores them in an array), and outputs these numbers in the same order as they were input.
For example, the program should ask how many numbers are going to be input, and dynamically allocate memory for an array to accommodate these numbers. The numbers should then be read one by one at a new line, e.g.:

```
Enter number: 34
Enter number: 1003
```

etc.
and then output in the correct order, e.g. `You have entered the numbers: 34, 1003, …,` etc.

You may want to make use of functions that read lines from the input, e.g. look at `getline.`

Exercise 4.

Pointers to functions are pointers, i.e. variables, which point to the address of a function. A running program gets a certain space in the memory. Both the executable compiled program code and the used variables, are put inside this memory. Thus a function in the program code is like, e.g. a character field, nothing else than an address. A pointer to that function will point to the address in the memory where that function is held.
The syntax is really awkward:

```
void Function_Pointer(float a, float b, float (*pt2Func)(float, float))
```

`pt2Func` is a pointer to a function that takes as parameters two variables of type `float`, and also returns a result of type `float`.
Important: A function pointer always points to a function with a specific signature. So all functions, you want to use with the same function pointer **must have the same parameters and return-type**.

As an exercise:
Implement a simple program that chooses which of the four basic arithmetic operations to execute at run time. There are other ways to implement this, but for now we need to use pointers to functions.

- Implement four functions that return `float` and take as parameters two variables of type `float`. These functions implement the four basic arithmetic operations, `add, subtract, multiply, divide`.
- Then implement a simple function:
  `Operation_Selection(float a, float b, float (*pt2Func)(float, float))`
  that simply returns as a result the outcome of the arithmetic operation function that pt2Func points to.
- Implement a simple `main` function that drives the program. Inside this function, for example, you can call `Operation_Selection` directly with any of the four operations.
- Insert appropriate output commands to check the output of the program at various stages.