LW: Passing by Reference

For completion credit, you need 25 out of 40 points.

Objectives

- To understand that:
  - argument passing is similar to initialization,
  - when a function is called, each formal argument is initialized by its corresponding actual argument, and
  - when a reference is provided as a formal parameter, the corresponding actual argument will be referred to by the formal parameter.
  - references are, at their fundamental core, pointers to the original memory, which is why changes persist after execution of the function.

Overview

You may have noticed that when calling a function with an argument, that changing the argument within the function will (generally) not affect the value of the argument passed in after execution of the function. Any change to the value inside of the function will not persist after the function returns. This behavior is referred to as **pass-by-value**, which makes a **copy** of the arguments and modifies those copies. Essentially, in pass-by-value the variable within the function is initialized with the value passed into the function.

In contrast, **pass-by-reference** does not copy the argument; it actually hands the function the original memory associated with the argument. In C++, we default to pass-by-value, but we can tell the compiler to use pass-by-reference by appending an ampersand (`&`) to the data type. For example, to pass an integer `'foo'` by reference, we would place int `&` foo in the argument list. In pass-by-reference the function uses the exact same variable as the one passed in, except it now might have a different name or nickname.

C++ also inherits from the C programming language the ability to **pass-by-reference** by using a pointer type (e.g. `int *k`). We then get a pointer to a variable at the call site using `&`. (e.g. if we have int a, we can pass a to the `int *` parameter as `&a`). Inside the function we must explicitly dereference the pointer (e.g. `*k`).

Your goal for this labwork is to successfully implement four functions which will be described below. Discuss the concepts with your group to solidify your understanding of pass-by-reference.

Requirements

- Navigate to Canvas, and download the starting source code.
    - Three files can be found in the starter.`zip` : `functions.h`, `functions.cpp`, and `main.cpp`
        - `functions.h` contains the prototypes of the functions. You will need to modify this file.
        - `functions.cpp` contains the implementations of the functions. You will need to modify this file.
        - `main.cpp` contains code to help you test the results of your functions. You will need to modify this file to call `function_three`.
- You should not need any additional #include statements. In particular, we are not intending you to use the C++ array or vector classes for function_four.
- Implement **`function_one`** and its prototype in `functions.h` so that it takes two integer arguments i and j, adds 2 to i, adds 1 to j, but after execution of the function, only j is changed.
    - Example: if `i = 3`, and `j = 4` and I call `function_one(i,j)` then after execution, `i = 3`, and `j = 5`.
- Implement **`function_two`** and its prototype in `functions.h` so that it takes an argument of type `struct example`, and increments its integer by 1. This change should persist after the function.
    - The structure `'example'` is defined in `functions.h`.
    - Example: if I have a structure named foo that is of type example, with `foo.value = 4` and I call `function_two(foo)` then after execution, `foo.value` should be `5`.
- Implement **`function_three`** with the given arguments so that it increments both variables by one. Both changes must persist after the function call.
    - **NOTE: DO NOT MODIFY THE FUNCTION ARGUMENTS for `function_three`**
        - `k` is a pointer to a single integer, and is not intended to be an array.
        - Example: if `a = 3`, and `b = 4` and I call function_three with the parameters a and b (appropriately passed) then after execution, `a = 4`, and `b = 5`.
- Implement **`function_four`** so that it takes in four arguments: an integer array, an integer size that contains the size of the array, and two integers called lowest and highest.
    - Iterate through the array, increment every value by two, and then set the arguments lowest and highest to the lowest and highest values in the array. The changes to the array, lowest, and highest should persist after execution.
    - Example: if
        - `lowest = 0`, `highest = 0`, `size = 5`, and `arr = {0,1,2,3,4}`,

      then after calling `function_four(arr, size, lowest,highest)` we should see

        - `arr = {2,3,4,5,6}`, `smallest = 2`, and `highest = 6`.