EE1305, Lab 6: Functions and parameter passing

Note: additional info added in green below for PGM challenge

Motivation

In this lab you will get started writing functions and passing parameters by value and by reference.

In each of the exercises below, we've given you the main program, and you'll need to write one or more C++ functions to complete the program. You'll need to determine what the parameter list (inputs and output parameters) of the function is, create a function prototype, and write the function itself.

Background information:

1. Parameter passing in C++

   a. Pass-by-Value vs Pass-by-Reference
   In C++ you can pass arguments to functions either as copies of the original variable (pass-by-value), or as references to the original variable (pass-by-reference). Value-parameters, copies of the original variable, can be changed without affecting the original variable. They are useful to pass inputs to functions. Reference parameters are useful when you need the function to be able to modify the original argument. They are useful as outputs from a function or for parameters which are both inputs and outputs of the function. See sample programs 1 (sample_value.cpp) & 2 (sample_reference.cpp).

   b. Functions and arrays
   Arrays in C++ are always passed to functions by reference. This primarily an effeciency issue – it would take a lot of time to copy an entire array to pass to a function, while it is trivial to simply pass it the address of the array. When passing arrays to functions, keep in mind that C++ arrays are not very smart -- they only know where they begin (their starting address in memory), not how long they are. Thus when passing arrays to functions you generally need to pass both the array and a size. See

sample program 3 (sample_array.cpp). Sometimes you need to pass an array to a function which does not modify the array, and it is useful to be able to specify that the array won't be modified. Since C++ arrays are always passed by reference, C++ provides the keyword **const** which can be added to an array parameter to signify that the array is an input to the function and will not be modified by the function. See sample program 4 (sample_const_array.cpp).

c. Returning more than one value from a function
If a function has only a single output (a string, an int, etc.), it is easiest to implement this with a value-returning function (see sample program 1). If a function has more than one output, it is customary to write a void function (a function whose return type is void) and have it pass its results back through two or more reference parameters (see sample program 2).

2. Function Prototypes:

The C++ compiler needs to know what a function looks like before it can compile any call to the function. Thus you either need to locate your functions in the file before they are called or insert a function prototype. Function prototypes look like a function header, but are terminated with a semicolon instead of being followed by the body of the function:

```
     int my_func(float param1, string
param2);       // an example of a
function prototype
```

Using function prototypes (rather than placing the function at the top of the program) is generally the best solution, because it allows for more flexibility on where functions are placed in the code. See sample programs 1, 2 & 3 above for examples of function prototypes.

(Many larger programs have their various functions split over several source code files, and each file will have an associated header (.h) file containing the prototypes for functions found in that file. Any source code file needing to use the functions found in a different file can simply `#include` the associated header file. Each file is compiled independently, and then after all modules are compiled, they are linked together in a separate step of the compile process to create a single executable program.)

3. Recursion

   Many problems can be solved elegantly with functions that call themselves recursively to get the job done. Tasks such as finding the shortest or fastest route (think google maps or the GPS system in a car, or an airline ticketing system) lend themselves nicely to a recursive solution. Other tasks, such instructions to a robot to move a pile of boxes (aka the Towers of Hanoi problem) are also nicely solved using recursion. One nice example is calculating x^n . x^n is $x * x^{n-1}$. A typical recursive routine has one or more base cases (x^0 is 1, x^1 is x), and a recursive call that moves the problem in the direction of a base case. Here's an example solving x^n .

4. Math needed in this lab:

   rectangular and polar coordinates

   1. 2-D rectangular coordinates (and vectors) are the usual cartesian coordinates (vectors) given as an x & y value pair.
   2. magnitude of a vector: given a 2-D vector **v** = (x,y), the magnitude of **v**, |**v**|, is simply $\sqrt{(x^2 + y^2)}$
   3. Polar coordinates represent a coordinate or vector as a magnitude and an angle. A given vector **v** = (x,y) can be given in polar coordinates **p** = (r, theta) where r = |**v**|, and theta is the arctangent of y/x. Note that in C++ there are 2 arctangent functions, `atan(s)` and `atan2(y,x)`, where s = y/x . Atan returns an angle between -PI/2 and PI/2, while atan2 returns an angle between -PI and PI. atan( )'s result is ambiguous – the vector from the origin to (1,1) and from the origin to (-1,-1) have the same return value (PI/4) from `atan( )`, while `atan2( )` distinguishes them (PI/4 for (1,1) and 5*PI/4 for (-1,-1)). The best way to calculate the angle theta is with:

      ```
      theta = atan2(y,x);
      ```

5. C++ Random Numbers:

   C++ provides a function which produces pseudo-random numbers: rand( ) . The numbers are random in the sense that there is no statistical correlation between one number and the next. They are 'pseudo'-random because the sequence is repeatable, providing you start with the same seed value. (The author describes random number generation in chapter 3 – be sure you review that section of the chapter.) (The default seed for both Visual Studio's C++ compiler and for the Gnu C++ compiler is 1). To get a random number from the random number generator, you can use:

```
int my_num = rand( );          // get a
random number from RNG
```

To seed the RNG and get 10 random number, you can use:

```
unsigned int my_seed = 100;    // seed the
RNG.
int my_nums[10];
for (int i=0; i<10; i++) my_nums[i] = rand(
);
```

Note that if you don't call srand( ) to seed the RNG, the default seed value is 1; this means if you rerun your program, you'll get the same random sequence each time. Here is a sample random-number program that demonstrates using rand( ) and srand( ) to generate random sequences. It also shows how to seed the RNG from the system clock, which provides a reasonably random seed.

Examples:

1. Sample 1: Pass-by-value and value-returning functions.
2. Sample 2: Pass-by-reference and returning more than one value.
3. Sample 3: passing arrays to functions.
4. Sample 4: passing arrays by const reference.
5. Sample 5: generating random numbers with rand( ) and srand( )

Part 1a: Value-Returning Function and Prototype.

See example 1 for an example that may help you solve this problem

Complete this program, by adding a function called mag( ) to calculate the magnitude of a vector. You'll need to add a prototype before main( ), and the implementation (header & body) of the function after main( );

reminder: mag = sqrt $(x^2 + y^2)$

Calling it with something like:

```
vec_length = mag(x, y);
```

Name your program functions1.cpp

Part 1b: Another Function and Prototype

See example 2 for examples that may help you solve this problem

Complete this program, by adding your mag( ) function from lab 05, and add another function `rect2polar` (and prototype) that returns the polar form of a rectangular coordinate pair. The data type of the inputs, `x, y` should be double, as should the data type of the outputs `r` and `theta`. Since you already have a function which gets the magnitude of a vector, your `rect2polar(...)` function can call it to get half the job done. The commented code (which you'll need to un-comment) shows how your function should be called. Name your program pass_by_ref.cpp .

Reminder: The rectangular-to-polar function should take two arguments as input (x, y) and return two values (r, ɵ) where:

> r = mag (x,y)
>
> ɵ = atan2(y,x)
>
> note: rectangular coordinate (1,1), is equivalent to polar coordinate (1.414, PI/4); (-2, -2) should give you (2.818, 5PI/4)
>
> note 2: since your function returns its two values through the reference parameters, the return type of the function should be `void` .

Name your program pass_by_ref.cpp

Part 2a: Calculating the Maximum and Minimum of an Array

See example 3, 4 & 5 for examples that may help you solve this problem

Before you start, study and understand example 4 above. Then uncomment the commented-out line in the function `display_it(...)`

and recompile. This will demonstrate to you the purpose of the keyword `const`.

Complete this program: add two functions (& their prototypes) that return the maximum and minimum value of an array of x elements. The program should ask the user for these x elements.

> Hint: the functions will return a single integer value, and should have 2 parameters: the array (passed as a constant array), and the numbers of values in the array.
>
> Question: do the `min_element(...)` and `max_element(...)` functions modify the arrays in any way? If not, then the array parameter (in both the function prototype and the function definition) should be declared `"const"`. While your functions will work just fine without the keyword `const`, it is better to use it here. That signals to readers of your code that the array will not be modified by the function, and would trigger a compiler error if you tried to modify the array inside the function. This is one of C++'s features that helps reduce human error.

Name your program minmax_array.cpp

Part 2b: Filling an array in a function instead of in main

Move the task of querying the user for the array values to a function query_for_array(...) .

> Question: what should be the return type of this function?
> Question 2: Do you need the keyword const here?

Part 2c: One function to do several tasks (needs pass-by-reference)

Add a function array_average(int array, int length); that returns the average value of the elements in the array.
Then add another function array_stats(...) that returns the min, max and average of an array.

Useful hint: you already have functions to get the min, max and average. Your stats function can simply call them...no need to cut and paste to get the smaller jobs done!
Question: what should be the return type of this function? (hint: with more than one piece of information to pass back, your function should use all reference parameters, and not pass anything back via the return statement.)

Reminder: be sure to put ampersands (&) in front of your parameter names to make them reference parameters!

Part 2d: Using rand( ) to generate random numbers

Add code to your program from parts 2a-2c to fill the array `rand_array` with 25 values using rand( ), and . Rerun your program, to demonstrate that you get the same sequence of random numbers each time. Finally, add code to seed the RNG (random number generator) from the system clock, and run you program several times to verify that you get a different sequence each time. The sample program 5 above should help with both using the RNG and seeding it.

Question: (Same question as for part 2a)

Name your program minmax_array_random.cpp

Part 3: Recursion

Complete this program by adding a recursive function to calculate n-factorial (n!) . Recall that n! is n * (n-1) * (n-2) * ... 2 * 1. Note that a special case is that 0! is defined as 1. Be sure to check that your answers are correct. You should find they are wrong for values of n > 12; they will be for the looping version, which uses an int. Explain why that is in the comments at the top of the program.

Name your program lab_06_factorial.cpp

Optional Challenge: with PGM files and arrays

Complete this program by adding three functions and their prototypes. This program should read a PGM from a file, save it to a new file, and print the image header and pixel values to the console. You'll need one function to open a PGM file, a second one to save the PGM file already read into new PGM file, and the third one to display the array to the console.

(If the image is large, printing all the pixels may get rather awkward. If the image is large (say, more than 200 pixels total), you can prompt the user for a starting row and column and a number of rows and columns to print, then just display a selected portion of the array. This is optional, but you'll find this helpful when we get to the projects, so is worth the effort to do it now.)

Hints:

> The readPGM function should take as an input parameter the input file name. Question: what are its outputs? How will you get that information back to the calling code?
> The savePGM and printPGM functions should receive all the information needed (type (P2), # of cols, etc., as well as the pixel array. Since they will not modify the array, how should you pass the array in? How should you pass in the other parameters?

Name your program PGM_input_output.cpp

Note: If you are using a version of VS earlier than 2013, you will probably need to add the following line to your code:

```
#pragma comment(linker, "/STACK:16777216")
```

Why? This increases the stack size to 16 MB. Default stack size in earlier versions of VS was only 1 MB, and the 512x512 int array uses 1 MB. BTW, dynamic memory allocation and pointers is the right way to do this, but we're not there yet.

Deliverables:

Final program (main and functions) for parts 1b, 2c, and if you took on the challenge, part 3. Submit these on Blackboard. Turn in:

1. `pass_by_ref.cpp`
2. `minmax_array_random.cpp`
3. `factorial.cpp`

If you do the Challenge for the week, submit that as well.

4. `PGM_input_output.cpp` (optional)

Due date: Midnight, Tuesday, March 31, 2015