# CSE 425 Studio: Functional Programming I

These studio exercises are intended to give you experience with the basic functional programming features available in C++ as well as to begin exploring different ways in which they may be used.

In this studio you will again work in self-selected groups of ~3 people, and will report the results of your work on the studio exercises in an e-mail to the course account. Students who are more familiar with the material are encouraged to partner with those who are less familiar with it, and asking questions of your classmates and professor during the studio sessions is highly encouraged as well.

Please record your answers as you work through the following exercises. After you have finished please email your answers to the required exercises, and to any of the enrichment exercises you completed, to the cse425@seas.wustl.edu course account, with subject line "Functional Programming Studio I". The enrichment exercises are optional but are a good way to dig into the material a little deeper, especially if you breeze through the required ones.

PART I: REQUIRED EXERCISES

1. Form a team of ~3 people of your choice (preferably a mix of people familiar and unfamiliar with C++) and write down the names of the team members as the answer to this first exercise.

2. Create a new Visual Studio 2013 project for this studio (for example, named **`FunctionalStudioI`** or something similar that identifies which studio this is for). Modify the signature of the main function in your project's main source file (e.g., **`FunctionalStudioI.cpp`**) so that it matches the standard portable (e.g., between Windows and Linux) main function signature (and variable names) for a C++ program: **`int main (int argc, char * argv[])`**.

Declare and define a function that takes no parameters and has a void return type, and prints out the string **`"hello, world!"`** to the standard output stream, and call that function from within your program's main function. Build and run your program, and as the answer to this exercise please show your code and the output your program produced.

3. Abstract the function you wrote in the previous exercise so that it now takes a single parameter that is a reference to a **`const`** (C++-style) **`string`**, and prints it out to the standard output stream. In your main function, call it with the string **`"hello, world!"`** and also try calling it with other strings. Build and run your program, confirm that it behaves as it did in the previous exercise when passed the string **`"hello, world!"`** and confirm that it behaves as you would expect when passed other strings. As the answer to this exercise, please show your code and the output your program produced with the new version of this function.

4. In your main function, declare a (STL) list of (C++ style) strings, and push back all the strings with which you called the function in the previous exercise, into that list. Call the (STL) **`for_each`** algorithm with iterators to the beginning and (just past the) end of the list, and a lambda expression (with an empty capture list and a formal parameter of type **`constr string &`**) that prints each string to the standard output stream (the optional LLM text book has a good example of this on page 391). Build and run your program, and confirm that it behaves as in the previous exercise. As the answer to this exercise, please show your code and the output your program produced.

5. Outside your main function (i.e., within the global scope), declare two **`string`** variables, one for a left-delimiting string (e.g., something like **`"[`**) and one for a right-delimiting string (e.g., something like **`"]`**). Without changing its capture list, try using those variables in your lambda expression, e.g., printing them out before and after (respectively) each string that is passed as a formal parameter to the lambda expression. Build and run your program and as the answer to this exercise please explain what happened when you did that, and what your observations tell you about the scoping of global variables with respect to their use in lambda expressions.

6. Move the two delimiting string variables from the previous exercise from the global scope into your program's main function (i.e., make them local stack variables). Try to build your program, and observe what happens - then add those variables to your lambda expression's capture list, and then build and run your program. Build and run your program and as the answer to this exercise please explain what happened when you did that, and what your observations tell you about the scoping of local variables with respect to their use in lambda expressions.

PART II: ENRICHMENT EXERCISES (optional, feel free to skip or to try variations that interest you)

7. Use the (STL) **`for_each`** algorithm and a lambda expression that concatenates strings from the list onto an initially empty global **`string`** variable, and after that algorithm has run print out the resulting global string. Build and run your program, and as the answer to this exercise please show your code and the program's output.

8. Try different variations of making the string into which the other strings are concatenated global vs. local, and also try different variations involving global vs. local delimiter strings, etc. Build and run your program for each of these variations, and as the answer to this exercise please describe what you observed when you did that.