CSE 425 Studio: Control Flow II

These studio exercises are intended to extend your familiarity with ideas and techniques for iterative and recursive program control flow, again in a multi-paradigm programming language (C++). 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 and help those who are less familiar with it, and asking questions of your classmates and professor during the studio sessions is highly encouraged as well. There are also links on the main course web page to helpful web sites for these exercises, which you are also encouraged to use as resources. Please record your answers as you work through the following exercises. After you have finished please e-mail your answers to the required exercises, and to any of the enrichment exercises you completed, to the cse425@seas.wustl.edu account, with the subject line "Control Flow Studio II". The enrichment exercise is a chance to dig deeper into the material, especially if you breeze through the required ones.

PART I: REQUIRED EXERCISES

1. Form a team of ~3 people and write down your names as the answer to this exercise.

2. **Order of Evaluation.** Open up Microsoft Visual Studio 2013 and create a new project for this studio (for example, named `ControlFlowStudioII` or something similar that identifies which studio this is for). Modify the main function signature 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[])`.

Outside your main function, declare and define a function called `get_value` that uses `cout` and `cin` to print out a prompt asking for input from the user, reads in a single integer value, and returns it. Also declare and define another function called `get_product` that calls `get_value` twice, multiplies the two values it obtains, and returns the result. In your main function, use both `get_value` and `get_product` in different expressions, and output the results of those expressions. Build and run your program, and notice how many times you were prompted for input. As the answer to this exercise describe what you observed, and what those observations say about the evaluation order with respect to functions used within expressions in C++.

3. **Iteration.** Comment out the code from the previous exercises, and in your main function write a loop that iterates forever, prompting the user for an odd number and reading in an integer value each time. If the user inputs an even number the loop should repeat and prompt again, or if an odd number is input the loop should exit and then reach the last statement of the main function which should return 0. Build and run your program and input several even numbers and then an odd number (at which point your program should end). As the answer to this exercise, please show your code and the output it produced.

4. **Recursion.** Repeat the previous exercise but implement it using recursion rather than iteration. Build and run your code with the same sequence of inputs as in the previous exercise, and as the answer to this exercise please show your code and the output it produced.

5. **Observing non-determinacy.** This exercise and the next one are designed to explore a particular form of non-determinacy in which the C++11 `std::async` function template is called with a callable entity (e.g., a function, lambda expression, function object, etc.) and the arguments to pass to it, which it then can either dispatch immediately, dispatch later, or (as long as no other code requires a result from it) even defer indefinitely (potentially never running it). If a dispatch does occur, it may be either run synchronously or asynchronously (e.g., by a separate thread launched implicitly from the library code).

Write a simple function that takes an `unsigned long long` integer, computes its factorial (either iteratively or recursively), and prints out the result it obtained. Include the C++11`<future>` library and in your main function use `std::async` to dispatch several asynchronous calls to that function with different values of its parameter (so that each call will take a different amount of time, but watch out for overflow of the result if you use too large an initial value), as in:

```
std::async(my_factorial, x);

std::async(my_factorial, y);

std::async(my_factorial, z);
```

Try dispatching the different calls in different orders. Also try adding a sleep statement in the main function (and varying the duration of the sleep) after all of the calls to `std::async` have been made, so that if the dispatches are asynchronous the main thread of execution is kept alive for longer or shorter intervals relative to the completion of the recursive function calls, as in:

```
std::this_thread::sleep_for(std::chrono::milliseconds(1000));
```

Build and run your program and as the answer to this exercise please describe any non-determinacy you saw in the order of completion of the tasks (or possibly even whether or not they are even run) relative to the order in which they were dispatched.

6. **Working with (or around) non-determinacy.** Repeat the previous exercise but have the recursive function return its result by value after it prints it out. Modify your main function so that it stores a future object for the result of each asynchronous dispatch, as in:

```
std::future<unsigned int> answer_for_x = std::async(my_factorial, x);

std::future<unsigned int> answer_for_y = std::async(my_factorial, y);

std::future<unsigned int> answer_for_z = std::async(my_factorial, z);
```

Then, after all of the dispatches have been made, the main function should call the `get()` method of each future object to obtain and print out the result of its corresponding computation. Build and run your program with the statements involving the futures being sequenced in different orders relative to the order of their corresponding dispatching statements, and as the answer to this exercise please show the code and the output the program produced, and describe any interesting variations in behavior you noticed, either relative to the previous exercise or with different variations within this experiment.

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

7. Try building and running the code from the two previous exercises with another compiler and platform (e.g., g++ on shell.cec.wustl.edu). As the answer to this exercise please describe any differences you noticed in the program's behavior, between the Windows studio environment and that one.

8. Implement one of the recursive functions shown on pages 273 or 274 of the Scott book, using C++ rather than Scheme. Build and run your code and as the answer to this exercise please show your code and the output your program produced.