Second Studio Session on C++ Algorithms

These studio exercises are intended to give additional coverage of the C++ Algorithms Library, and to give you experience using it within the Visual C++ environment.

As before, students who are more familiar with the material are encouraged to help those for whom it is less familiar. Asking questions of your professor and teaching assistant (as well as of each other) during the studio sessions is highly encouraged as well.

Please record your answers you work through the following exercises. After you have finished, please submit your answers to the required exercises and to any of the enrichment exercises you completed to Blackboard or our OJ system (if applicable).

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. Open Visual Studio and create a new Visual C++ Empty Project for this studio. In your main function, declare variables of each of the following parameterized sequence container types: **`deque<int>`**, **`list<int>`**, **`forward_list<int>`**, and **`vector<int>`**. Push back the same set of five unique integer values into each of these containers, and then for each of the containers write code that positions an iterator two positions from the start of the container's range and uses the iterator to print out the value at that position, in the most efficient manner possible.

   For random access containers this can be done in constant time using pointer arithmetic expressions directly on the iterators returned by the container's **`begin()`** and **`end()`** methods, but for other containers you will get an error message if you try to use pointer arithmetic (try this), and instead you must use other operators . As the answer to this exercise, explain how you were able to accomplish this in the most efficient manner, for each of the different containers' specific kind of iterator.

2. Ranges are defined from the first iterator position given, up to but not including the second iterator position given. Think about how you would represent an empty range using two iterators, given that assumption for how ranges are defined. Using iterators that point to different positions in your containers and use them with the copy algorithm (and an output iterator as in the previous studio) to print out different empty and non-empty sub-ranges of each of the containers, with the values in each range all on the same line with spaces in between them. As the answer to this exercise, please explain what happens when you do that, and show your program's output.

3. In your main function, again declare a (plain-old-C++-style) array of integers, initialized with an odd number of unsorted values in which each value appears one or more times. For example:

   ```
   int arr[] = {-2, 19, 80, -47, 80, 80, -2};
   ```

As in the previous studio, use the **`copy`** algorithm to print out the contents of the array by passing it the starting address of the array, the pointer that points just past the end of the array, and a variable of type **`ostream_iterator<int>`** (an output iterator for an **`ostream`**) that is initialized with **`cout`**. Then, have your program sort the contents of the array into non-increasing order (largest to smallest) by passing an object of type **`greater<int>`** (you'll need to include the **`<functional>`** library) as a third parameter to the sort algorithm and again use the **`copy`** algorithm to print out the contents of the array after that sort. As the answer to this exercise, show the output your program produced before and after the array was sorted.

4. Repeat the previous exercise, but instead of passing **`greater<int>`** to the algorithm, write your own (plain old C-style) function (that is not a member of a struct or class) that takes two integers by value and returns a **`bool`** value that is true if and only if the first value is greater than the second one, and pass that function to the algorithm instead. Build and run your program and make sure that the output it produces is the same as it was in the previous exercise. As the answer to this exercise, please show the code that you wrote in order to achieve that.

5. Repeat the previous exercise, but instead of passing a function to the algorithm, write your own struct that has an overloaded **`operator()`** (the function call operator) that takes two integers by value and returns a **`bool`** value that is true if and only if the first value is greater than the second one. Declare an instance of that struct type and pass that instance to the algorithm instead of the function from the previous exercise. Build and run your program and make sure that the output it produces is the same as it was in the previous exercise. As the answer to this exercise, please show the code that you wrote in order to achieve that.

PART II: ENRICHMENT EXERCISES (optional, feel free to do the ones that interest you).

6. Repeat any of the previous exercises that involves a callable object, using a C++11 lambda expression instead of a function, function object, or function pointer.

7. Take any callable object that is a binary predicate for the (strictly) greater-than relation and adapt it using the bind function to be a unary operator that returns true if and only the value it is passed is greater than 0. Use the adapted callable object with one or more of the C++ STL algorithms to print out all of the positive (non-negative, non-zero) integers in the array from exercise 3.