CSE 332 Second Studio Session on C++ Templates and Generic Programming

These studio exercises are intended to introduce more advanced features of C++ templates and to give you experience using them within the Visual C++ environment. In this studio you will again work in small groups. 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 instructors and teaching assistants (as well as of each other) during studio sessions is highly encouraged as well.

Please record your answers you work through the following exercises. After you have finished please send your answers to the required exercises, and to any of the enrichment exercises you completed, in an e-mail to the cse332@seas.wustl.edu course e-mail account, with the subject line "Templates Studio II". 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.

Please make sure that as you work through these exercises that each member of your team has a chance to participate actively – one way is to take turns coding, looking up details, debugging, etc.

PART I: REQUIRED EXERCISES

1. Find your team members in the studio area, sit down at/around and log in to one of the Windows machines, open up Visual Studio, and create a new Visual C++ Win32 Console Application project for this studio. Change the signature of the main function in the source file that Visual C++ generated to match the one that was specified in the lab assignments and in the lecture slides. Write down the names of the team members who are present (please catch up anyone arriving late on the work, and also add their name) as the answer to this exercise.

2. Add a new header file to your project and in that file provide a function template definition with a single **`ITERATOR`** type parameter (for the type of iterator the function will use) that takes two arguments of the parameterized iterator type and returns a **`size_t`** value containing the number of elements in the range (from the position pointed to by the first iterator, up to but not including the position pointed to by the second iterator). Implement this definition by returning the result of subtracting the first iterator from the second one. In your main function, declare an array of integers and a list of integers (including any necessary library header files, etc.) and add different numbers of elements to them. In each case print out the result from calling the counting function using appropriate iterators: pointers to the start of the array and to just past the last element in the array in the first case, or the list's **`begin()`** and **`end()`** methods in the second case. If either of these cases cannot be made to compile, just comment it out and get the other one to work (hint: one of them should work and one of them shouldn't). As the answer to this exercise, please describe (a) what happens and (b) why you think it does that, in each of those cases.

3. In that same file, above your original definition of the function template, provide another separate definition of the same function template that will work for both the list and array iterators (hint: you can move a list iterator forward with the ++ operator). Complete the next part of this exercise before attempting to build your code.

   Modify the signature of your original definition (which relies on subtracting the arguments passed to the function) so that instead of taking arguments of the parameterized type, that function template is **overloaded** to take pointers to the parameterized type as its arguments. Add a line to each of the definitions so that it prints out a unique message (so you can see which one is being called when your program is run).

   Again call the counting function using appropriate iterators: pointers to the start of the array and to just past the last element in the array in the first case, or the iterators returned by the list's **`begin()`** and **`end()`** methods in the second case (hint: now both of them should work). Build and run your program, and as the answer to this exercise, please describe what happens (and why you think it does that) in each of those cases.

4. Certain operators and functions must be available for the parameterized types (since otherwise the code will not compile). As the answer to this exercise, for each of the function template definitions you developed in the previous exercise, please list **all** of the methods (including constructors) and operators that must supported for the parameterized type in order for that definition to work with it.

5. Fully specialize the function template for pointers to integers: it should have an empty template parameter list and should take two pointers to integers as its arguments. In addition to computing and returning the absolute value of the distance between the passed pointers, this definition should print out all of the integers in the range on a single line, with spaces separating them. As the answer to this exercise describe how the output of your program changed between exercise 3 and this one.

6. Add another full specialization of the function template for pointers to characters: it should have an empty template parameter list and should take two pointers to characters as its arguments. In addition to computing and returning the absolute value of the distance between the passed pointers, this definition should print out all of the characters in the range on a single line, with no intervening spaces.

   In your main function declare and fill in an array of characters and print out the result from calling the counting function using pointers to the start of the array and to just past the last element in the array. As the answer to this exercise, describe how the output of your program changed between the previous exercise and this one.

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

7. As the answer to this exercise, please respond to each of the following questions: (1) Which additional methods and/or operators that must be supported for the argument types in exercises 5 and 6 beyond those required in the previous exercises? (2) Is the set of type requirements imposed by one of the definitions in the previous exercises more strict than in any other definition (and if so in what way)? (3) How many of the definitions would be able to work (e.g., if other more specific definitions were not provided) with pointers to characters? (4) How many of the definitions would be able to work (e.g., if other more specific definitions were not provided) with pointers to double precision floating point numbers?

8. Code up the print function template examples from the lecture slides, and as the answer to this question describe any challenges you ran into while doing that, as well as any observations you may have about how those templates work and how other types we've not considered would work with them.