CSE 332 Studio Session on C++ Standard Template Library Functors

Functors unify the notion of a function and the notion of an object, so that functions can be used as objects (copied, passed around, etc.) and objects can be used as functions (invoked, their results obtained, etc.). These studio exercises are intended to introduce information and techniques for C++ Standard Template Library (STL) functors, and to give you experience using them in the Visual C++ environment. More details about functors can be found using the Visual Studio help utility and the useful web pages at http://www.cppreference.com/wiki/stl/functional/ and http://www.sgi.com/tech/stl/table_of_contents.html

In this studio you will again work in small groups and as before students who are more familiar with the material are encouraged to help those for whom it is less familiar, and asking questions during the studio sessions is highly encouraged as well. Please record your answers as you work through the following exercises. When finished please post your answers to the required exercises, and to any enrichment exercises you completed, to the course message board as a reply to my posting titled “STL Functors Studio”. The enrichment exercises are an optional but good way to dig into the material a little deeper, especially if you breeze through the required ones. Please make sure as you work through these exercises that each team member has a chance to participate actively – e.g., take turns coding, looking up details, debugging, etc., and please also refer to the slides and the posted code examples as you work.

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. Change the signature of the main function in the source file that Visual C++ generated to match the one that was specified for the previous studios. 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. Declare a container of type vector<size_t> and push back the values 2 through 100 into it. Write a struct template with a const operator() (the function call operator) that takes a single value of the parameterized type and returns a bool indicating whether or not that value is prime (only evenly divisible by itself and 1). Use the STL find_if algorithm and an instance of that struct template to go through the vector and print out all of the prime numbers found in the integers ranging from 2 through 100. Build your program and fix any warnings or errors you encounter. Run your program and give the list of primes it printed out, as the answer to this exercise.
3. Write a struct template with a single type parameter that is derived through public inheritance from the STL binary_function template, with both argument types and the return type of the base class template parameterized with your struct template’s parameterized type. In your struct template define a const function call operator that takes two arguments of its parameterized type by value. This operator should returns a value of the parameterized type that is the result of raising the first argument to the power of the second argument (**hints**: while it is possible to use the C library pow function to do this as is illustrated by the slides on STL functors in the course management system, that approach can run into issues with type conversion – it’s easy and more general to implement this function call operator with a loop that does repeated multiplication; also, if you forget to make your function call operator const you can run into compiler errors – for example it may say something about const-volatile properties).

Call the STL transform algorithm with begin and end iterators for the vector from the previous exercise, an appropriate ostream iterator, and the result of using the STL bind2nd functor adapter to bind the value 2 as the second argument of an object of your struct template type (so that the resulting object is a unary functor that computes the square of the number passed to it). Build your program, fix any errors and warnings, and run your program which should now output the squares of the numbers in the vector from the previous exercise. Call the transform algorithm again but this time bind the value 3 instead of 2 (so that the resulting object is a unary functor that computes the cube of the number passed to it). Build your program, fix any errors and warnings, and run your program which should now output the cubes as well as the squares of the numbers in the vector from the previous exercise. As the answer to this exercise, show the list of squares and the list of cubes, of the numbers from 2 through 100.

4. Use the STL partition algorithm and the struct template from exercise 3 to partition the primes from the non-primes, in the vector from the previous exercise. Use the STL copy algorithm, the iterator returned by the partition algorithm, and an appropriate ostream iterator to print out of the primes on one line, and the non-primes on another line. Build and run your program. Notice that the partition algorithm does not preserve the order in which the primes and non-primes were found originally. Use the STL sort algorithm on the appropriate ranges to put the primes into ascending order and the non-primes into ascending order, before printing out those lines. As the answer to this exercise, say which numbers from 2 to 100 were prime and which were non-prime.

5. Declare a (plain-old-native-C++) array and initialize it with C-style strings of different lengths, as in: const char * words[] = {“hello”, “to”, “the”, “vast”, “world”}; Declare a container of type vector<size_t> and then call the STL transform algorithm using the appropriate iterators for the array and the vector (**hint**: if necessary, look at the previous studio’s exercises for how to use a back_insert_iterator) to push the sizes each of the strings from the array back into the vector, using the C library strlen function as the functor that converts a C-style string into its length. Use the copy algorithm with appropriate ostream_iterator objects to print the contents of both containers, and as the answer to this exercise, give the output that this produced.
6. Declare a container of type `vector<string>` and use the copy algorithm with the appropriate iterators to push the C-style strings from the array in the previous exercise, back into the vector you just declared. Use the `copy` algorithm and an `ostream_iterator` of `string` to print out the vector of C++-style strings.

Then, use the STL `transform` algorithm to copy the contents of the vector into the same type of ostream iterator that worked for printing the array of C-style strings in the previous exercise. To do this, create a functor that can convert a C++ style string into a C-style string, by passing a pointer to the `c_str` member function of the `string` class to `mem_fun_ref` (hint: use the address-of operator on the scoped member function name to get such a pointer-to-member-function). As the answer to this exercise, please show the output produced for both cases (which should be the same).

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

7. Declare a struct template with two type parameters and a function call operator that takes an object of the first parameterized type by value, uses that object to construct a dynamically allocated object of the second parameterized type, and returns a pointer to the dynamically allocated object. Also declare a struct template with a single type parameter and a function call operator that takes a pointer to an object of the parameterized type (by value), calls delete on that pointer, and has a void return type.

Use these functors to repeat the previous exercise, but this time (1) use a `vector<string *>` instead of a `vector<string>`, (2) use transform and your functor that does dynamic allocation to convert pointers to char into pointers to (dynamically allocated) strings, (3) use `mem_fun` instead of `mem_fun_ref` to turn `string::c_str` into a suitable functor when printing the contents of the vector, and (4) after that use the `for_each` algorithm and your functor that calls delete to destroy the dynamically allocated strings. As the answer to this exercise, please show your definitions of the two functors, and explain briefly how each one is called and what it does in the code.

8. Repeat exercise 3, but use the STL `transform` algorithm with the STL `plus` and `multiplies` binary function templates, your exponentiation functor, the STL `bind1st` and/or `bind2nd` functor adapters, additional vectors to store intermediate results, and appropriate back insert iterators for those vectors, to compute and print out the result of evaluating the expression $3x^2 + 1$ using the values in the original vector for $x$ in each case. As the answer to this exercise, show your code that did this.