**MSCF Financial Computing II**

**Homework 4**

***Due At 12:59 pm Monday, Nov. 19, Before the Midterm***

***You will lose 10 points per hour after that time***

1. **(100 points) Containers, Iterators, Algorithms, and Distributions**
2. Create a new, empty project named **FCII\_HW4**. In Source Files, add an existing item, **FCII\_HW4.cpp**. This source code file defines and inserts values into two containers, a **vector<int> v1** and a **multimap<char,int> mm1**, from random integers between 0 and 9, and random letters between ‘a’ and ‘e’. The code that inserts values into **mm1** illustrates how to declare and use a **pair<char,int>** object. **Because rand() returns values in the range [0,32767] in Visual Studio, we have “skipped ahead” to part of Lecture 17, and are using a *random number engine* from <random>. minstd\_rand uses a *linear congruential* algorithm that produces “minimum standard” random integers in the range [0,2147483647]. For our purposes, this is “like” rand() but with a wider range of values.** Examine the code, then compile and test, to confirm that the program works as expected.
3. In part (b) of **FCII\_HW4.cpp**, uncomment and examine the given code. The **time()** function in **<ctime>** returns the number of seconds that have elapsed since the start of Jan. 1, 1970. We have used **time()** here as a good-enough timer for creation and search of a large **vector<int> v3**. The first half of the code creates a vector of 50,000,000 random **int** values (using **msrand()**), and reports the time needed to do this; you will see that **push\_back()** is *very* efficient. The second half of the code uses the **find()** algorithm in a loop to perform 1000 linear searches of **v3**; you will see that this is *very* slow. The **find()** algorithm returns an iterator for the found value, or *container***.end()** if the value is not found.

***In Visual Studio, compiling in 32-bit Debug mode will make things much slower than they need to be. Change from* Debug *to* Release*, and from* x86 *to* x64*.***

Compile and test, and confirm that the results make sense.

1. In part (c) of **FCII\_HW4.cpp**, copy and modify the code from part (b), as follows:
   1. Instead of creating a new **vector<int>** variable, use the **sort()** algorithm to sort the elements of the existing **vector<int> v3**. Report the time needed for this.
   2. Instead of using **find()** to perform a linear search in **v3**, use the **binary\_search()** algorithm to take advantage of **v3** being in sorted order.
   3. Instead of performing just 1000 searches, perform 10,000,000. Report the time needed for this.

Compile and test, and confirm that the results make sense.

1. In part (d) of **FCII\_HW4.cpp**, copy and modify the code from part (c), as follows:
   1. Instead of creating a **vector<int>** variable, create a **set<int> s2** containing 50,000,000 random **int** values. You will need to use **set<int>::insert()** rather than **vector<int>::push\_back()** for this. Report the time needed to create **s2**.
   2. Instead of using the **find()** algorithm to search for a value in **s2**, use the **set<int>::find()** member function, which is optimized for sets.
   3. Instead of performing just 1000 searches, perform 10,000,000. Report the time needed for this.

Compile and test, and confirm that the results make sense.

1. In part (e) of **FCII\_HW4.cpp**, copy and modify the code from part (d) above, as follows:
   1. Instead of creating a **set<int>** variable, create an **unordered\_set<int> us2** containing 50,000,000 random **int** values. Use **unordered\_set<int>::insert()** for this, which has the same interface as **set<int>::insert()**. Report the time needed to create **us2**.
   2. Use the **unordered\_set<int>::find()** member function for lookups, which has the same interface as **set<int>::find()**.
   3. Report the time needed for 10,000,000 lookups.

Compile and test, and confirm that the results make sense.

1. Part (f) of **FCII\_HW4.cpp** provides code that creates a **vector<double> dv1** of the size of **v1**, initialized with all **0.0**s by default. Then, the square roots of the elements of **v1** are computed using the **sqrt()** function from **<cmath>**, and stored in the corresponding elements of **dv1**. Uncomment the code in part (f) and compile and test, to confirm that it works as expected.

Then: (i) use the correct form of **vector<T>::assign()** to assign all **0.0**s to the elements of **dv1**; (ii) use the **transform()** algorithm rather than a **for** loop to compute the square roots of the elements of **v1** and store the results in the corresponding elements of **dv1**; and, (iii) display the elements of **dv1** following **transform()**. ***Hint:*** The “obvious” thing to do is:

**transform(v1.begin(), v1.end(), dv1.begin(), sqrt);**

But **sqrt()** is *overloaded*—**float sqrt(float)**, **double sqrt(double)**, **long double sqrt(long double)**—and the compiler will not be able to figure out which version to call. We need to use a *cast*, to tell the compiler which version of **sqrt()** to call:

**transform(v1.begin(), v1.end(), dv1.begin(), (double (\*)(int)) sqrt);**

Compile and test, to confirm this works as expected.

1. In part (g) of **FCII\_HW4.cpp**:
   1. Above **main()**, define a function **add\_rand()** that returns **void** and that takes a reference-to-**double** as its parameter. **add\_rand()** should modify its parameter by adding a random value to it, from the range [0.01,1.00].
   2. In place of the comment for part (g), define a **vector<double> dv2** as a copy of **dv1**. Display **dv2** to confirm the copy.
   3. Next, write a **for\_each()** algorithm that calls **add\_rand()** on each element of **dv2**. Display the modified **dv2** to confirm that the **for\_each()** algorithm succeeded.

Compile and test, and confirm that the results make sense.

1. In part (h) of **FCII\_HW4.cpp**:
   1. Use the **random\_shuffle()** algorithm to shuffle the values in **dv2**, then display **dv2**.
   2. Use the **reverse()** algorithm to reverse the values in **dv2**, then display **dv2**.
   3. Use the **sort()** algorithm to sort the values in **dv2**, then display **dv2**.
   4. Use the **lower\_bound()** algorithm to find the lower bound of **2.0** in **dv2**.
   5. Use **lower\_bound()** again to find the lower bound of **3.0** in **dv2**.
   6. Use a **for** loop to display all the values in **dv2** between the lower bound of **2.0** and the lower bound of **3.0**.

Compile and test, and confirm that the results make sense.

1. In part (i) of **FCII\_HW4.cpp**:
   1. Use **transform()** to change each element *x* in **dv2** to **exp(***x***)**, then display **dv2**.
   2. Use **transform()** to change each element *x* in **dv2** to **ceil(***x***)**, then display **dv2**.
   3. Use **transform()** to change each element *x* in **dv2** to **log(***x***)**, then display **dv2**.

Compile and test, and confirm that the results make sense.

1. A second form of **transform()** combines the elements of *two* containers (which could be the same container twice) using a *binary* operation (a two-argument function), with the results stored into an output container (which could be one of the two input containers). The interface to this version of **transform()** looks like:

**transform(***beg1***,** *end1***,** *beg2***,** *begout***,** *binop***)**

where *beg1*, *end1* are the beginning and end of a sequence from one input container, *beg2*

is the beginning of a sequence from the second input container (which must be at least as

long as the sequence from the first), *begout* is the beginning of a sequence in the output

container (which must be long enough), and *binop* is the binary operation.

For example, **pow(***x***,***y***)** from **<cmath>** is a binary operation that can be used with this

version of **transform()**.

In part (j) of **FCII\_HW4.cpp** create an output **vector<double> dv3** of the same size as **dv1** and **dv2**, then use this version of **transform()** to store **pow(dv1[***i***], dv2[***i***])** into each **dv3[***i***]**, then display dv3. Compile and test, and confirm the results make sense.

1. In part (k) of **FCII\_HW4.cpp**:
   1. Declare a **vector<int>** object **v2**, initialized as a copy of **v1**. Use the **equal()** algorithm to test whether the **vector<int>** objects **v1** and **v2** are equal. Display the return value from the **equal()** algorithm.
   2. Do a **random\_shuffle()** on **v1**, and use **equal()** again; display the return value.
   3. Use the **is\_permutation()** algorithm to test whether **v1** is a permutation of **v2**, that is, has the same values but perhaps in a different order; display the return value.

Compile and test, and confirm that the results make sense.

1. In part (l) of **FCII\_HW4.cpp**:
   1. Use the **count()** algorithm to count the number of **3**s in **v1**; display the return value.
   2. Use the **minmax\_element()** algorithm to locate the minimum and maximum values in **v1**; display these values, using the pair of iterators returned by **minmax\_element()**.
   3. Use the **find()** algorithm to find the first **9** in **v2**; display the values in **v2** from that **9** to the end of **v2**.

Compile and test, and confirm that the results make sense.

1. Containers **vector<T>**, **deque<T>**, **list<T>**, and **forward\_list<T>** have a **resize()** member function that can be used to shrink or grow the container. Use **resize()** to cut the size of **v1** in half, then display the modified **v1**, concluding with newline. Compile and test, and confirm that the results make sense.
2. The order of an associative container (**set**, **multiset**, **map**, **multimap**) can be modified through specification of a comparison operator type as an additional template argument. The default ordering is **less<T>** where **T** is the element type. At the end of **main()**, define these two **set**-of-**int** objects:

**set<int> s1;**

**set<int, greater<int>> s2;**

(You will notice that the **set** definition requires the ***type*** of the comparison, **greater<int>**, whereas **transform()** requires an ***object*** to do the comparison, **greater<int>()**, which invokes the default constructor to create the object.) Use **copy()** and a (general) inserter to store the values of **v1** into **s1** (use **s1.begin()** as the “hint” initial position). Do the same again, but this time into **s2**. Display the elements in **s1** and in **s2**. Compile and test. You should see that **s1** and **s2** contain the same values, but in opposite order.

1. Here is example code illustrating one million random draws from N(0,2) using **mt19937**, the 32-bit Mersenne Twister random number engine. **mte()** returns a random integer; **nd(mte)** returns a random draw from N(0,2); **round(nd(dre))** rounds the random draw to the nearest integer value, but as a **double**; **int(round(nd(dre)))** converts this value to an **int** for use as a key/subscript in the counting map.

**mt19937 mte;**

**normal\_distribution<> nd(0.0,2.0);**

**map<int,int> buckets;**

**for (int i(0); i < 1000000; ++i) {**

**buckets[int(round(nd(mte)))] += 1;**

**}**

**for (auto& el : buckets) cout << el.first << "\t"**

**<< el.second << '\n';**

Compile and test, and confirm that the output makes sense.

1. Copy and modify the code from part (o) to display an empirical CDF (cumulative distribution function) of your sampled normal distribution. For example, if we had only drawn 100 samples and got this output in (o):

**-6 1**

**-5 3**

**-4 4**

**-3 6**

**-2 8**

**-1 15**

**0 19**

**1 18**

**2 14**

**3 8**

**4 3**

**6 1**

Then our new output in (p) including the empirical CDF would be:

**-6 1 0.01**

**-5 3 0.04**

**-4 4 0.08**

**-3 6 0.14**

**-2 8 0.22**

**-1 15 0.37**

**0 19 0.56**

**1 18 0.74**

**2 14 0.88**

**3 8 0.96**

**4 3 0.99**

**6 1 1**

Run a test with 1,000,000 draws from N(0,1), a standard normal distribution, including the CDF. (You can continue using **mte** from (o) as your random number engine, or you can define a new random number engine object.) Compile and test. How does your CDF compare with what you would expect from N(0,1)?

1. Create a test of 1,000,000 draws from a uniform distribution of integers in the range 1 through 10, inclusive. Display a table similar to the one in (p), showing the integer, the number of draws of that integer, and the CDF up to that integer. Compile and test.

***REMEMBER*** to put all authors’ names into your source code file.Put your **FCII\_HW4.cpp** file into a **.zip** archive and upload to the course web site.