CISC 2000—Computer Science II
Fall, 2013

Programming Project #5: Timing Has Come Today[1]
Date Due: Monday 9 December 2013

(Based on Exercise 20.20 in the text.)
We have seen that the C++ Standard Template Library (STL) provides

- a number of useful containers,
- iterators, a uniform interface for accessing containers, and
- a number of useful algorithms.

We have also mentioned that different containers are optimized for different operations. For example, insertion of one element at a known point is faster for a `list` than it is for a `vector`, but accessing an arbitrary element is faster for a `vector` than it is for a `list`.

In this assignment, you will be asked to determine the time it takes to fill a container with a given number of randomly-generated data values. Your code will do this for several different container types, namely,

- `list`,
- `vector`, and
- `set`.

In addition, for the `list` and `vector` containers,[2] you are to insert the data in increasing order.

By comparing the amount of time it takes to fill these containers, you can get a feel for which might be most efficient. For example, it might well be the case that one container is better for small data sets, whereas another might be better for large data sets. But we won't know unless we run these tests.

Of course, we'll want some evidence that once we've filled the `list` and the `vector` with data, that this data is actually sorted. Since you're going to be testing some very large data sets, printing out the contents and then "eyeballing" the results is not a viable option. The good news here is that the C++11 `stl` provides an `is_sorted()` algorithm. Its declaration is

```
template <class Iter> bool is_sorted(Iter first, Iter last);
```

This function returns `true` if the range `[first,last)` is sorted into ascending order.[3] The bad news is that `std_lib_facilities.h` doesn't work with C++11. This means that you'll need to include the following headers near the beginning of your program:

[1] With apologies to the Chambers Brothers

[2] The `set` container is automatically sorted. So you don't need to do anything special to keep the data sorted, and there's no point in checking whether the data *is* sorted, when you're using a `set`.

[3] In case you were wondering, this function uses <, as defined for the type of elements stored in the container, to determine sortedness.

```
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <list>
#include <set>
#include <vector>

using namespace std;
```

The overall structure of your code (top-level pseudocode) will look like the following:

```
int main()
{
   srand(time(0));              // initialize random number generator

   prompt for input
   while (cin >> num_elts) {
      if (num_elts <= 0)
         print error message
      else {
         vector<double> data = gen_data(num_elts);
         time the insertion from data into a list
         time the insertion from data into a set
         time the insertion from data into a vector
      }
      prompt for next input
   }
}
```

It's probably clear that this assignment is supposed to give you experience using the STL. However, it requires you to know how to generate random data, as well as how to time a code segment. Since we haven't covered these topics yet, I will give you this information for free.

For generating random data (see Chapter 24) and storing it in a vector, use the following function:

```
// generate num_elts random numbers in the range [0.0, 1.0),
// which are returned in a vector
vector<double> gen_data(int num_elts)
{
   vector<double> result;
   for (int i = 0; i < num_elts; i++) {
      double datum = 1.0*rand()/RAND_MAX;
      result.push_back(datum);
   }
   return result;
}
```

The way one does timings is as follows (see Chapter 26):

```
clock_t t1 = clock();
if (t1 == clock_t(-1)) {    // clock_t(-1) means "clock() didn't work"
   cerr << "sorry, no clock\n";
   exit(1);
}

// insert code that needs to be timed here

clock_t t2 = clock();
if (t2 == clock_t(-1)) {
   cerr << "sorry, clock overflow\n";
   exit(2);
}

cout << "Elapsed time: " << fixed << setprecision(2)
     << static_cast<double>(t2 - t1)/CLOCKS_PER_SEC << " seconds\n";
```

At this point, it should be clear that you can solve the problem in two ways. On the one hand, you can write three nearly-identical chunks of code that "fill in the blanks" for the timing code fragment given above. This is tedious and error-prone; moroever, you would probably tear out your hair if I were to ask you to throw in an additional data type (such as `unordered_set`) at the last minute. On the other hand, you could write functions that do data-insertion for each of the data types. Then your `main()` function would call these functions within its loop. I'll let you figure out which technique is better, keeping in mind that programming style is an important ideal, not to mention a big part of your grade.

I found it useful to first define an `Inserter` as a synonym for a the kind of function that would take the already-prepared data (which was stored in a `vector<double>`) and insert it into a container, i.e.,

```
typedef void Inserter(vector<double>);
```

Next, I wrote a function having prototype

```
void time_insert(Inserter inserter, vector<double> data);
```

that would do the work in the timing block given above, i.e., it would find the starting time, do the work of inserting the data (in sorted order), find the stopping time, and then report the elapsed time. Having done so, I could then write functions such as

```
void insert_list(vector<double> data);
```

Since `time_insert` is an example of an `Inserter`, you can put

```
time_insert(insert_list, data);
```

within `main()`. The other containers would be handled analogously by functions `insert_set()` and `insert_vector()`.

A few considerations:

1. Do all your work in the directory `~/private/cs2/proj5` (this should come as no surprise).

2. Use the facilities of the STL as much as possible. For example, since we're trying to keep the container sorted at all times, you'll find that `find_if` is *very* helpful for finding the spot where each new data item should be inserted. **This won't apply to inserting data into a `set`, since a `set` is automatically sorted.** However, `find_if` needs a predicate that specifies the "finding criterion"; you'll find that the `Larger_than` template class (found in Chapter 21) to be useful here.

3. The directory `~agw/class/cs2/share/proj5` has the following goodies:

   - A `Makefile`, which you should copy to your working directory. It works in the usual way; see previous project handouts (or the brief documentation at the top of the `Makefile`) for details.
   - A working version of `proj5`. Run it in the usual way; see previous project handouts for details. You should try it out on containers having the following sizes: -1, 0, 1, 10, 100, 1000, 5000, 10000, 50000. This will give you a feeling for how long things should take to run.

Deliverables: Please turn in a *clean* typescript consisting of

- a listing of `proj5.cc`,
- evidence of correct compilation (e.g., `make clean` followed by `make`),
- a sample run of `proj5`. Use the following values for your container size: -1, 0, 1, 10, 100, 1000, 5000, 10000, 50000. Be prepared to wait a while for these larger containers.

Send it to me via email, in the usual way; see previous project handouts for details.

Good luck!