Advanced C++ Lab 1: Diving into the Standard Library

Reading

Take some time to browse the online STL documentation graciously provided by SGI (10 years ago!). This guide will be your very good friend this term. You should especially become familiar with the Table of Contents, which lets you find documentation of concepts and implementations very quickly, and the Index, which lets you track down any odds and ends you might need for your programs.

Practice

The Standard Template Library provides a great set of efficient, generic, modular tools for writing programs. Try the following warm-up exercises in STL. Save each of the following code snippets in a file named `exercise`**`n`**`.cc`, where **n** is the exercise number.

1. Create a vector of 100 `int`s. Fill it with 100 small, random integers between 1 and 100, inclusive, by using the appropriate mutating algorithm. Generate these numbers by passing the mutating algorithm a function object (or perhaps a simple function) that models a `Generator`. (Your function object should use `rand()` and the modulo operator, at least.)

   Since it would be nice to verify your code is working, use the `copy` algorithm to write your vector to `cout` via an `ostream_iterator` parameterized on `int`s.

2. Create a new vector to hold 100 `string`s. Use another `Generator` that you write to fill it with 100 random alphabetic strings of random lengths between 5 and 15 characters inclusive.

   Sort the vector you've made using the version of `sort` that applies `operator<` to the vector. Display this sorted vector in a similar fashion to the way you displayed exercise 1's result.

   Then, re-sort the vector by length of string (ascending). You'll want to use a comparator that conforms to the Strict Weak Ordering model. Output your re-sorted vector to stdout.

3. Generate a vector of 100 random integers between 1 and 100, as before, then write some code to count how many of these numbers are odd and how many are even, in a single pass over the integers. Do this by creating a simple function object that implements the `unary_function` model, and that maintains some internal state. You should then be able to apply it with the `for_each` algorithm and then extract the counts at the end.

4. Have a look at this code:

   ```
   vector<int> v;
   v.push_back(1);
   v.push_back(4);
   v.push_back(2);
   v.push_back(8);
   v.push_back(5);
   v.push_back(7);

   copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
   cout << endl;

   vector<int>::iterator new_end =
           remove_if(v.begin(), v.end(),
                     compose1(bind2nd(equal_to<int>(), 0),
                              bind2nd(modulus<int>(), 2)));

   copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
   cout << endl;
   ```

   (The `compose1` function can be difficult to track down in the `g++` compiler. It should be in the `__gnu_cxx` namespace, and you will need to `#include <ext/functional>` to get the definition.)

   Paste it into `exercise4.cc`, and use a comment block at the top to explain:

   1. what you think the author of this code intended it to do,
   2. why this code is broken (what *does* it do?),
   3. how you've fixed it.

   Of course, actually fix the code, compile, and verify that it works the way it was probably meant to work.

   Use the STL docs to help you understand what `bind2nd` and `compose1` do.

   - (Hint 1: They're easy!)
   - (Hint 2: Finding them in the STL documentation *isn't* easy! See the Introduction section for Function Objects.)

5. Now you can try your hand at writing a simple STL algorithm. You will create your own version of the `reverse()` algorithm in a file `exercise5.cc`. This algorithm must be implemented as a *function template* (which we will discuss in a future class), declared like this:

```
template <typename BidirectionalIterator>
void my_reverse(BidirectionalIterator first, BidirectionalIterator last) {
    // Your implementation here!
}
```

Of course, the arguments `first` and `last` can be assumed to model the BidirectionalIterator concept; they support ++ and --. These arguments specify the range of values `[first, last)` to reverse; i.e. with a `vector<int> v`, you could reverse the vector's contents with the statement `my_reverse(v.begin(), v.end())`.

The STL provides a couple of helper functions to simplify your life:

- The `swap(Assignable &a, Assignable &b)` function allows you to swap two values (hint: `swap(*first, *last)`).
- The `distance(InputIterator first, InputIterator last)` function tells you the distance between two iterators.

You can look in the SGI STL documentation to find out what STL headers these functions are declared in.

Create a helper function `test(int size)` that does the following:

- Output a message to console indicating the specified size, e.g. "Size: 20".
- Create a vector of integers with `size` elements
- Populate this vector's elements with random values in the range [1..100]
- Output the vector's contents to the console, with elements separated by a space
- Reverse the vector with your `my_reverse` function
- Output the vector's contents again

Then, in your `main()` function, call your test-function with an even size and an odd size (e.g. 20 and 17), so that you can make sure your reverse function works properly. You should also test with sizes 0 and 1 to make sure your function handles those situations.

Submission

We will figure out the submission mechanism and notify you of how to submit your files before HW1 is due. Thank you for your patience!