Programming in C^{++}, Exercise List 8

Deadline: 28.04.2015

In this exercise, we study `std::map< >` and `std::unordered_map< >`. They have similar functionality: Each of the two versions of `map<X,Y>` implements a table of elements (x, y) with $x \in X$ and $y \in Y$, in such a way that y can be efficiently looked up, when x is known. One could also say that `map<X,Y>` implements a lookup table from X to Y.

The difference between `std::map<X,Y>` and `std::unordered_map<X,Y>` is the mechanism that is used for lookup: `std::map< >` uses a search tree, so that it requires an order X. `std::unordered_map< >` is based on hashing, so it needs a hash function and an equality function on X.

1. Write a function

   ```
   std::map< std::string, unsigned int > frequencytable(
            const std::vector< std::string > & text )
   ```

   that constructs a table of frequencies of the words in `text`.

   Inserting into a map can be tricky when Y has no default constructor, but in this task you can simply use `[ ]`. In a later exercise, we will treat `[ ]` in more detail, because it has some problems with constness of the map and default construction of elements of Y.

2. Write a function

   ```
   std::ostream&
   operator << ( std::ostream& stream,
                 const std::map< std::string, unsigned int > & freq )
   ```

   that prints the frequency table. Use a **range-for**.

3. `std::map< >` uses by default the order `<` on `std::string`. We want the frequence table to be case insenstive. Try for example:

   ```
   std::cout << frequencies( std::vector< std::string >
      { "AA", "aA", "Aa", "this", "THIS" } );
   ```

   In order to overcome this problem, we will have to provide our own comparator. Define a class

```
struct case_insensitive
{
   bool operator( ) ( const std::string& s1, const std::string& s2 ) const;
      // Return true if s1 < s2, ignoring case of the letters.
};
```

Class `case_insensitive` has only one constructor, namely its default constructor. Test it for example by

```
case_insensitive c;
std::cout << c( "a", "A" ) << c( "a","b" ) << c( "A", "b" ) << "\n";
```

There is no `==`-operator. `std::map` will assume that two objects are equal when both `c(s1,s2)` and `(s2,s1)` are false.

Write `bool operator( )` in a reasonable fashion! Making a lower case copy of the string, and using `<` is not reasonable.

4. Once you have finished the `case_insensitive` class, you can do one of the following things, dependent of your level of eagerness:

   - Simply replace `std::map< std::string, unsigned int >` by `std::map< std::string, unsigned int, case_insensitive >`, in everything that you wrote before, and sorting should work as desired.
   - Make `operator <<` and `frequencytable` polymorphic: Write:

     ```
     template< typename C = std::less< std::string >>
     std::map< std::string, unsigned int, C >
     frequencytable( const std::vector< std::string > & text )
        // frequencytable( test ) will produce a frequency table, using
        // std::less< std::string >, which is the good old <
        // frequencytable<case_insensitive>( test ) will produce a
        // case insensitive frequency table.

     template< typename C >
     std::ostream& operator << ( std::ostream& out,
                   const std::map< std::string, unsigned int, C > & m );
        // Prints frequencytable, for every possible comparator.
     ```

5. Now we want to write the same functions with `std::unordered_map`. If we will do nothing, comparison will also be case sensitive here, so we need to create a case-insensitive hash function, and a case-insensitive equality function. They work in the same way as the `case_insensitive` object:

   ```
   struct case_insensitive_hash
   {
      size_t operator ( ) ( const std::string& s ) const
   };
   ```

```
struct case_insensitive_equality
{
   bool operator ( ) ( const std::string& s1,
                       const std::string& s2 ) const
};

case_insensitive_hash h;
std::cout << h( "xxx" ) << " " << h( "XXX" ) << "\n";
std::cout << h( "Abc" ) << " " << h( "abC" ) << "\n";
   // Hash value should be case insensitive.

case_insensitive_equality e;
   std::cout << e( "xxx", "XXX" ) << "\n";
      // Prints '1'.
```

6. If everything went well, the following function is now easy to write:

```
   std::unordered_map< std::string, unsigned int >
    hashed_frequencytable(
       std::vector<std::string> ( { "aa", "AA", "bb", "BB" } );
          // As with frequencytable for map, you can make this
          // function polymorphic.
          // The default parameters are std::hash<std::string> and
          // std::equal_to<std::string>.
```