CSCI 235, Programming Languages, C^{++}
Exercise 5

Deadline: 24/26.09.2018 (day of your assigned lab)

This eclectic exercise covers may topics at the same time: Usage of `std::list< >` and `std::vector< >`, file handling, use of namespaces, use of input parameters, and time measuring. Namespaces are a convenient way of avoiding name conflicts in big programs. Our program will be not big, but we need to get used to using them.

Download the files **listtest.h**, **listtest.cpp**, **vectortest.h**, **vectortest.cpp**, **nr06.cpp**, **timer.h** and the **Makefile** from Moodle. You can use `std::string` or your own string class.

1. Complete the function

   ```
   std::vector< std::string >
   vectortest::readfile( std::istream& input )
   ```

   in file **vectortest.cpp**. This function reads from `input` and creates a vector containing all words in `input`. This function should ignore everything that is not a letter (uppercase or lowercase). The returned vector must never contain empty strings. Characters can be recognized by `int isalpha( int )`, defined in `<cctype>`.

   `bool input.good( )` means that the last operation on `input` succeeded. It does not mean that the next operation will succeed. One must first read a character (as `int`), and then check if it was read correctly.

   Function **readfile** can be called by declaring `std::ifstream inp{ "filename-to-read-from" }` and using `inp` as argument.

   Use `inp.get( )` for reading a character from `inp`. Don't use `>>`.

   Inputfile containing

   ```
   46S hello, 4
   world,,,X
   ,,Y,y
   ```

   should result in vector `{ "S", "hello", "world", "X" }`. Test this function carefully! My experience with the labs is that a large group of students lacks elementary programming skills.

2. Complete the functions

   ```
   std::ostream&
   operator << ( std::ostream& , const std::vector< std::string > & );
   std::ostream&
   operator << ( std::ostream& , const std::list< std::string > & );
   ```

   in files **vectortest.cpp** and **listtest.cpp**. They are not in the namespace, because uniqueness is guaranteed by their type.

   The functions should print a `{`, the elements of the vector (or list), separated by commas (`,`) and ended by a `}`. Again, test this function carefully with length $0, 1, 2, 3$ at least. We are not going to compromise.

3. Add the following sorting functions to **vectortest.cpp**:

   ```
   void vectortest::sort_assign( std::vector< std::string > & v )
   {
      for( size_t j = 0; j < v. size( ); ++ j )
         for( size_t i = 0; i < j; ++ i )
         {
            if( v[i] > v[j] )
            {
               std::string s = v[i];
               v[i] = v[j];
               v[j] = s;
            }
         }
   }

   void vectortest::sort_move( std::vector< std::string > & v )
   {
      for( size_t j = 0; j < v. size( ); ++ j )
      {
         for( size_t i = 0; i < j; ++ i )
         {
            if( v[i] > v[j] )
               std::swap( v[i], v[j] );
         }
      }
   }

   void vectortest::sort_std( std::vector< std::string > & v )
   {
      std::sort( v. begin( ), v. end( ));
   }
   ```

The first sorting function exchanges strings by usual assignment. The second sorting function uses `std::swap`, which swpas the strings by exchanging the pointers. The third function calls `std::sort`, which uses quicksort.

You may use cut-and-paste of course, but rearrange the lay out.

4. Systematically measure the performance of these sorting functions using input that is big enough. Use compiler optimization `-O3 -flto`.

   The best way to measure performance, is by using function `randomstrings( nr, s )`, which creates a vector of `nr` random strings of length `s`. Use a reasonably big `s`, e.g. 50. Use a `nr`, that gives reasonable times, (a few seconds). Measure for at least five different values of `nr`. Write a table.

   You can use a **timer**, defined in file **timer.h**. In order to use it, write

   ```
   { timer t( "some type of sorting", std::cout );
     ..... ;
   };   // Destructor measures and prints
        // time that t existed.
   ```

   Try to observe the following things:

   (a) Which sorting functions are $O(n^2)$, which are $O(n \cdot \log(n))$?

   (b) Among those with $O(n^2)$, which one is faster?

   (c) Is there any difference between unoptimized compilation and optimized compilation? How big is it on average?

5. Write the sorting functions that are declared in file **listtest.h**. Since `std::list` does not have indexing, you have to replace the indices by iterators. Unfortunately, `std::sort( )` cannot be used on `std::list`, because it requires random access. This means that there are only two sorting functions on `std::list`.

   Write a function that converts vectors of strings to lists of strings.

6. Measure the performance of the two sorting functions on `std::list`. What are the complexities? Which one is faster?

7. Finally, compare sorting on `std::list` with sorting on `std::vector`. Which is the fastest among `vectortest::sort_assign`, `vectortest::sort_move`, `listtest::sort_assign`, `listtest::sort_move`?

If you need documentation on `list` or `vector`, look at `http://www.cplusplus.com/`.