In this exercise we study STL iterators, algorithms and function objects.

Excercise 10 (Function objects, 1p)

Modify and improve the random number generator from the lecture slides to create a lotto program in the following ways:

First generate a set of 7 individual lotto numbers (in range 1 - 37) using the scheme shown at the lecture. Then print them. In the exercise 10 you used for loop to display numbers from the container. Do it now without a loop by using the output stream iterator.

After generating and printing lotto numbers, generate another, separate lotto number sequence and display it.

Then compare those lotto number sequences and print similar numbers (still without a loop, using for_each() algorithm instead) as a list (smallest first, here we have numbers 1, 11 and 24 in both of the lists):

```
First lotto row: 1 12 24 36 11 15 32
Second lotto row: 24 7 29 11 18 35 1
Same numbers:
#1: 1
#2: 11
#3: 24
```

Hint: set_intersection() algorithm may be useful to find out similar numbers.

Extra excercise 10b (STL-containers, iterators, algorithms and function objects, 0,25p)

Write a program that could be used in a sport event. Each athlete has a number, name and time.

In the beginning the names are read from the text file on the disk. Names are stored as lines so that name of each athlete consists of one line in the text file (use the given file athletes.txt, or create that kind of file using some text editor like notepad for example). The following code can be used to set the input stream to use '\n' as a field separator (instead of spaces), so that you can use spaces between names.

```
struct field_reader: ctype<char> { // specify '\n' as the only separator
      field_reader(): ctype<char>(get_table()) {}
      static ctype_base::mask const *get_table() {
            static vector<ctype_base::mask>
            rc(table_size, ctype_base::mask());
            rc['\n'] = ctype_base::space;
            return &rc[0];
      }
};
ifstream myfile("…filename…");
myfile.imbue(locale(locale(), new field_reader()));// use our own separator list
istream_iterator<string> iit(myfile);
```

When all the names have been read the numbers are drawn for each athlete. This is done so that the vector of athletes are shuffled (algorithm random_shuffle) and after the shuffle, the numbers are given in order 1, 2,…, n. The numbers are needed to be stored also in athlete objects so that they can be used later when athlete vector is sorted in ascending order according their time in the race.

The next step in the program is that times are entered from the keyboard in the order of their numbers. When this is done, the athlete vector is sorted in ascending order according the times.

The final step is that result list is displayed on the screen. The list contains number, name and time. Here is one output of the program run:

```
Give time for Cameron Diaz ? 23
Give time for Axel Richthofen ? 43
Give time for Raymond Banks ? 22
Give time for Isaskar Keturi ? 12
Give time for Victor Kulikov ? 76

Final results are:
3 Isaskar Keturi   12
4 Raymond Banks    22
5 Cameron Diaz     23
2 Axel Richthofen  43
1 Victor Kulikov   76
```

Remark 1. The times in running race can be expressed as one integer (seconds only).

Remark 2. It is required all the time that athlete vector is used as a whole via algorithms. So you must not write any loop in the program.

Remark 3. Start by writing the class Athlete that needs at least the following member functions: constructor, ReadName, ReadTime, Better (according the time comparison) and output operator <<.

Remark 4. Now there is a situation where member function of an object should be called from the algorithm for each object in the container. This is not possible directly, but the problem can be get around by writing the function that take the object as a parameter. In the implementation of this function the member function of the parameter object is then called.

The same problem could be solved in another way too. It is possible to create a function object whose constructor takes the address of the member function and stores it to it's data member. The function call operator function (operator()()) of this class takes the objects as

parameter. In the implementation of this function call operator the member function is then called via the member function pointer that was stored in the function object.

The advantage of the latter solution is that function object class can be written as a template so that container object type is a template parameter. And because member function pointer is passed as a parameter, this solution is very generic.

This kind of situation is very common when using STL. Because of that STL actually contains a template for these kind of function objects and it also has a helper function that further facilitates the use of the template so that function object for this purpose can be created simply this way: mem_fn(Class::member_func).)