Object-Oriented Programming in C++ (SS 2025)
Exercise 6: June 19, 2025

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Wolfgang.Schreiner@risc.jku.at

May 23, 2025

The exercise is to be submitted by the denoted deadline via the submission interface of the Moodle course as a single file in zip (`.zip`) or tarred gzip (`.tgz`) format which contains the following files:

- A PDF file `Exercise`*`Number`*`-`*`MatNr`*`.pdf` (where *Number* is the number of the exercise and *MatNr* is your "Matrikelnummer") which consists of the following parts:
  1. A decent cover page with the title of the course, the number of the exercise, and the author of the solution (identified by name, Matrikelnummer and email address).
  2. For every source file, a listing in a *fixed width font*, e.g. `Courier`, (such that indentations are appropriately preserved) and an appropriate *font size* such that source code lines to not break.
  3. A description of all tests performed (copies of program inputs and program outputs) explicitly highlighting, if some test produces an unexpected result.
  4. Any additional explanation you would like to give. In particular, if your solution has unwanted problems or bugs, please document these explicitly (you will get more credit for such solutions).
- Each source file of your solution (no object files or executables).

Please obey the coding style recommendations posted on the course site.

Exercise 6: Text Statistics with Containers

The goal of this exercise is to write a program that can be called from the command line as

```
statistics path n
```

where *path* denotes the location of a text file and n is a natural number. The program prints those n words that occur most often in the file together with the number of their occurrences. A word is a non-empty sequence of letters; a letter is a character for which the function `isalpha()` returns true[1]. All other characters are not part of a word but separate them; every character is mapped to its lower-case equivalent[2] before further processing.

The implementation of the program shall be based on classes that implement the following interface:

```
typedef tuple<string,int> Word;
class WordProcessor
{
public:
  virtual ~WordProcessor() {}
  virtual void add(string word) = 0;
  virtual int size() = 0;
  virtual void sort() = 0;
  virtual Word word(int i) = 0;
};
```

where `add()` enters a new word from the text and `size()` returns the number of different words encountered in the text. A call of `sort()` ensures that the words are sorted according to their rank (in descending order); any subsequent call of `word(`i`)` returns a tuple that contains a word and the number of its occurrences ($i = 0$ denotes the word with the largest number of occurrences, $i = 1$ the word with the second-largest number and so on; i must be less than the value of `size()`). Please note that `std::tuple` is a class template of the standard library defined in header `<tuple>` (lookup its definition).

First, write a class template

```
template<template<typename V, typename... R> class S>
class SeqWordProcessor: public WordProcessor
{ ... };
```

that implements the text processor with the help of a *sequence container* class template *S* that can be instantiated with a type *V* (where *R* represents any additional optional arguments that the template may have): the class template maintains a sequence of type *S*`<Word>`. If a word is entered, the sequence is searched for the word; if the word does not occur in the sequence, a new `Word` object is created, initialized with the word and occurrence 1 and added to the end of the sequence; if the word already occurs in the sequence, the number of occurrences is increased by one. A call of `sort()` sorts the sequence according to the number of occurrences of each word. Since *S* can be an arbitrary sequence container, rather than sorting the sequence in place, a call of `sort()` first generates a `vector` of the `Word` values of the sequence that is then sorted according to the number of occurrences; from this vector, subsequent calls of `word()` are handled.

[1] http://www.cplusplus.com/reference/cctype/isalpha
[2] http://www.cplusplus.com/reference/cctype/tolower

Next, implement a class template

```
template<template<typename K, typename V, typename... R> class A>
class AssocWordProcessor: public WordProcessor
{ ... };
```

that implements the text processor with the help of an *associative container* `A`: the class template maintains a map of type `A<string,Word>` that maps a word to the corresponding statistics information. The implementation proceeds in a similar way as described above except that instead of a search a map lookup takes place.

The program shall instantiate these templates to create text processors of type

```
SeqWordProcessor<vector>
SeqWordProcessor<list>
AssocWordProcessor<map>
```

For each text processor, the program shall read the file, enter the words, print the results and the number of their occurrences, and how long the total process took[3].

Use for your tests the text you can download from

http://www.gutenberg.org/files/1524/1524-0.txt

If the timings are to short go give accurate results, process the text m times and divide the time by m, for a suitable value of m. If the timings take much too long, use only a part of this file (and submit the truncated version of the file as part of the deliverable).

[3] http://www.cplusplus.com/reference/ctime/clock