Advanced Programming in C++

Standard library – implementing insertion techniques

This exercises is about inserting elements in a sequence container in different ways and with different techniques. Basically, strings shall be stored in a std::vector in lexicographical order, but together with each string also an **int** is to be stored. Use std::pair for storing such string/**int** elements in the vector. The **int** value is to be used for, either counting how many times a certain string has occurred in input, or for showing in which order a string was read from input. Input/output examples is given below. A skeleton program, skeleton.cc, is given in directory given_files. In this file you will find comments that describe what is common for all versions of the exercise:

- Declare an alias for std::pair, storing a std::string and an **int**.
- Declare an alias for std::vector, storing std::pairs as described above.
- Overload a function insert(*string, vector*) to insert a string, together with an **int**, as an std::pair in the vector. One version shall use copy semantics for inserting the string, the other move semantics.
- Overload **operator**<< for the pair type and the vector type, so std::ostream_iterator can be used to output the elements in the container when overloading **operator**<< for the container. You will typically experience ADL problems for one of these overloadings.

A small test data file, data.txt, is given in directory given_files. It contains nine short strings:

```
foo fie fum abc foo xyz fie foo fum
```

Several versions of the program shall be implemented. Elements shall always be stored in *alphabetic order* regarding the strings. *Two main cases* for storing elements shall be implemented:

1) If a string occurs several times in input, only *one* entry for that string shall be stored in the vector, and the **int** value shall in this case be used to *count* the number of occurrences. For the given file data.txt, the output for this case shall be as below.

```
abc (1)
fie (2)
foo (3)
fum (2)
xyz (1)
```

2) If a string occurs several times in input, one entry for *each* occurrence shall be stored in the vector, and the **int** value shall in this case show the order in which the string was read from input. A later occurrence of a string shall be inserted after the previous entries of the same string. For the given data file data.txt, the output shall be as below(“abc” occurs once in the given input file, as the fourth string; “fie” occurs twice, as the second and seventh string, e.g.):

```
abc (4)
fie (2)
fie (7)
foo (1)
foo (5)
foo (8)
fum (3)
fum (9)
xyz (6)
```

The exercise is to implement a number of variations of n case 1 and case 2, by using either algorithm find_if(), lower_bound() or upper_bound() to find the position in the vector for a certain string (find_if() have linear, while lower_bound() and upper_bound() have logarithmic time complexity). For inserting a string/**int** pair, either vector insert(), alternatively emplace(), shall be used, or vector push_back(), alternatively emplace_back(), in combination with algorithm rotate() (rotate is more useful than one might expect, and can be very efficient, *not* said that is the case here; both alternatives have linear time complexity).

When searching for a string, a comparison function must be supplied to the seach algorithm, try both function objects and lambda expressions. Always try to implement comparisons by using only < and ==, in combination with logical negation (!). For comparable types, < is the minimal set of relational operators on can expect, and == is the minimal set of equality operators.

To do

- Implement **case 1**. Use **find_if**() to find the appropriate position for a string in the vector. If the string is found, increment the count. If not found, use **insert**(), alternatively **emplace**(), to insert a new entry for that string with count set to 1.

  A solution is found on file find_if-insert.cc.

- Implement **case 2**. Use **find_if**() to find the appropriate position for a string in the vector. Use **insert**(), alternatively **emplace**(), to insert a new entry for that string in alphabetical, chronological order, where the integer value shows the order in which that string was read from input.

  A solution is found on file find_if-insert-multi.cc.

- Implement **case 1**. Use **lower_bound**() to find the appropriate position for a string in the vector. If the string is found, increment the count. If not found, use insert(), alternatively **emplace**(), to insert a new entry for that string, with count set to 1.

  A solution is found on file lower_bound-insert.cc.

- Implement **case 2**. Use **lower_bound**() to find the appropriate position for a string in the vector. Use **insert**(), alternatively **emplace**(), to insert a new entry for that string, in alphabetical, chronological order, where the integer value shows the order in which that string was read from input

  A solution is found on file lower_bound-insert-multi.cc.

- Implement **case 1**. Use **lower_bound**() to find the appropriate position for a string in the vector. If the string is found, increment the count. If not found, use **push_back**(), alternatively **emplace_back**(), to insert a new entry for that string, with count set to 1, at the end of the vector, and then use algorithm **rotate**() to rotate the element into place.

  A solution is found on file lower_bound-rotate.cc

- Implement **case 2**. Use **upper_bound**() to find the appropriate position for a string in the vector. If not found, use **push_back**(), alternatively **emplace_back**(), to insert a new entry, where the integer value shows the order in which that string was read from input, at the end of the vector, and then use algorithm **rotate**() to rotate the element into place (alphabetical, chronological order).

  A solution is found on file upper_bound-rotate-multi.cc.