Contest notes: "Admission problem"

S. Shershakov
G. Zhulikov

April 4, 2023

Ed. 1.4 as of 04.04.2023

1 Goals and Outcomes

Goals This assignment is mostly related to dealing with standard containers that manage objects of a custom datatype. Sorting issues are considered in the context of using predicates.

Outcomes Students successfully completing this assignment would master the following: sorting elements of custom datatypes stored in containers of various types.

2 Task description

The description is provided as it can be found in the definition section of the corresponding context problem. The description is written by another author.

Develop a program that distributes students to universities.

Each university has a limit of places, each applicant has a certain score and a list of preferences.

The distribution is as follows: applicants are sorted by score in descending order, when score is equal — by date of birth, surname and name in increasing order.

Then, each applicant goes to the first university from his list, which has vacant places left (if there are no such places, then the applicant goes nowhere).

2.1 Input format

The first line contains a single integer N from 1 to 10^4 — the number of universities.

The next N lines contain a string of Latin characters from 5 to 15 characters and a number from 0 to 10^9 — the name and the maximum number of students for the next university.

The next line contains an integer M from 0 to 10^4 — the number of applicants.

Next come the M lines, each of which contains two strings from 5 to 15 characters — the name and surname of the next applicant, then 3 integers from 0 to 10^9 — number, month and year of birth, an integer from 0 to 10^9 is a student's score, an integer k from 0 to 200 is the number of universities to which the applicant is ready to enroll, and k of university names.

Listing 1. Sample input

```
3
MSU 1
HSE 2
```

```
MIPT 100
5
Ivan Ivanov 1 1 1900 100 2 MSU HSE
Petr Petrov 2 1 1900 90 2 MSU HSE
Alexander Sidorov 3 1 1900 110 2 MIPT HSE
Ivan Petrov 3 1 1900 100 3 HSE MSU MIPT
Petr Ivanov 4 1 1900 80 1 HSE
```

2.2 Output format

For each university, print its name in alphabetical order, then, through tabulation, the first and last names of the students who entered it, sorted by surname, name and date of birth.

Listing 2. Corresponding output

```
HSE     Ivan Petrov     Petr Petrov
MIPT    Alexander Sidorov
MSU     Ivan Ivanov
```

3 General idea

In order to cope with this task, a proper decomposition of a problem/program is expected. At least, the following methods are suggested to be developed:

- a method for reading a collection of universities from a given input stream;
- a method for reading a collection of applicants from a given input stream;
- a method that sorts a collection of applicants according to their scores and additional information; this method is considered only if the collection of applicants is not auto-ordered;
- a method that distributes applicants according to their scores and preferences;
- a method that outputs a list of universities and related applicants.
  The `main()` method ties them all together and solves the problem.

3.1 Representing a sorted collection of applicants

The general approach to deal with the problem is rather straightforward. One needs to represent a collection of applicants sorted to according their scores. Then one needs to iterate the sorted collection and then, for each applicant, determines whether is it possible to enrol them to the first university chosen by them — if there exist at least one vacancy. Otherwise, we consider the second choice, then the third. If there is no vacancy available at any university, a student is not enrolled at all.

There are two individual approaches to deal with sorted collections. *First*, the collection is represented using some linear container such as `std::vector`. The initial data is added to such a container using the most appropriate way[1]. After that, the container is sorted by a sorting procedure. A `std::sort()` *algorithm* can be considered a good choice. *Second*, a self-sorting container

[1] For instance, `push_back()` for vectors.

is considered as a primary structure for adding and storing data from the very beginning. `std::set` and `std::map` are examples of such containers that allow to efficiently[2] *search*, *insert* and *remove* elements.

Further we consider both approaches, one for each part[3] of the application.

[2] All major operations are guaranteed to be logariphmic with regard to the number of elements.

[3] Conventionally, the first part is related to the data input, and the second part is related to the enrollment procedure.

4 Implementation details

We need to choose proper structures for storing a collection of universities and a collection of applicants. Universities are determined only by their names, so a simple `std::string` is enough to represent a single university. Each applicant is represented by a set of related attributes, so it will be convenient to define a custom datatype. A skeleton for a structure to represent applicants can be seen in the following Listing.

Listing 3. A skeleton for a structure representing an applicant.

```
struct Applicant {
    std::string name;
    ...
};
```

We need a few containers to store both universities and applicants[4]. For instance, we can use a map for storing together names of universities and number of vacancies. It is reccomended to create a type alias such as:

[4] Moreover, we can also consider additional auxiliary containers further.

```
typedef std::map<std::string, unsigned int> UniCountMap;
```

For storing a collection of applicants that are read from a stream, it is just enough to create a vector:

```
typedef std::vector<Applicant> ApplicantsVector;
```

4.1 Sorting a linear container

After filling this vector up with the applicants, we can simply sort it by the `std::sort()` method[5]. This method obtains 3 parameters: first two are the `[first, last)` range. The last one is a so-called *predicate* (or *comparator*[6]) `comp`. A *predicate* is an entity, for which the `operator()` can be applied.

[5] www.cplusplus.com/reference/algorithm/sort/

[6] See http://www.cplusplus.com/forum/general/102545/ for example.

We may pass a *pointer to a function* for such a predicate. For instance, let's consider a function that is defined as follows:

```
bool appPred(const Applicant& lhv, const Applicant& rhv)
{
    if(lhv.score > rhv.score)
        return true;
    else if(lhv.score < rhv.score)
        return false;

    if(lhv.surname < rhv.surname)
        ...
}
```

The function obtains exactly two parameters and returns *true* if a position of the first parameter is considered as being *before* the position of the second parameter; *false* otherwise. Having such a function and a vector of applicants `apps`, we may simply sort[7] this vector by using the following code:

[7] The Complexity of sorting of an unordered sequence is known to be *linearithmic*, namely $N \cdot log(N)$.

```
1 std::sort(std::begin(apps), std::end(apps), appPred);
```

After sorting, we just iterate through the vector and put applicants to an appropriate university.

4.2 Self-ordered containers

`std::set` and `std::map` are examples of containers that maintain their internal structure to be ordered. There are several approaches that allow doing so. *First*, a default predicate can be considered. It means that elements of a self-ordered container are sorted using `std::less<T>` predicate which compares two given elements of a collection using `operator<`. This approach is not very suitable for custom datatypes, because it requires overloading the `operator<`, which hardcodes the order for such a datatype. *Second*, a custom predicate based on a structure with overloaded `operator()` can be passed as a parameter of a structure datatype definition. This approach gives you flexibility for applying a different collation order by changing the predicate or using different predicates for different collections. *Third*, a function can be used as a custom predicated, similarly to how a function is used in `std::sort` in the previous section. This approach is not as flexible as the second one - you have to create a new collection to change the predicate - and it is not as clear.

Further, we consider the second approach. Let's consider two cases under which we have to sort a collection of students. At the first part of the application we need a collection of students sorted primarily by a *score* criterion. Then, if for two given students, they have the same scores, their names and birthdays must be also considered. At the second part of application, only the surname, the name and the birthday are considered while printing a student together with a university where the applicant is enrolled.

It means, that we can use almost the same predicate for both cases, but the *score* is considered only once. We can create a custom comparator using the following code skeleton:

```
1  struct ApplicantPred {
2      // defines whether to consider score while sorting or not
3      bool considerScore;     //                          Represents a comparator state.
4
5      ApplicantPred(bool consScore = false)
6          : considerScore(consScore)          //          By default we do NOT consider score, only
7      {                                                   names/bday are considered.
8      }
9
10     bool operator() (const Applicant& lhv, const Applicant& rhv)
11         const                                //          By definition, this method MUST be marked
12     {                                                   as const, because it does NOT change the
13                                                         instance's state.
```

```
14          if(considerScore)                   //
15          {
16              if(lhv.score > rhv.score)
17                  return true;
18              else if(lhv.score < rhv.score)
19                  return false;
20          }
21
22          if(lhv.surname < rhv.surname)
23              ...
24      }
25  };
```

Here we use the internal state of the comparator in order to determine whether we have to consider score or not.

By using such a comparator, we may remaster the sorting procedure of the vector of applicants as follows:

```
1  ApplicantPred pred(true);       //
2  std::sort(std::begin(apps), std::end(apps), pred);
```

Now we DO consider scores.

Then, we use the predicate as a parameter of a type definition of a self-ordered set:

```
1  typedef std::set<Applicant,         //
2                   ApplicantPred      //
3                  > ApplicantsSet;
```

Type of elements of the set.

Type of the comparator of the set. This is a type, not an object!

5 *Questions for self-study*

1. Why do we set `ApplicantPred::considerScore` to `false` by default? What will change if we set it to `true`?[8]
2. Compare the complexity (in terms of some elementary operations) of using 1) a vector followed by sorting and 2) a set. Use a reference[9] if needed.

[8] Try to consider a specific application of `ApplicantsSet`. For instance, how to represent a collection that links together a university and a set of students enrolled to the university?

[9] E.g. www.cplusplus.com/