DSBA Introduction to Programming // Midterm Programming test

Spring semester 2021/22

Date: Apr 18, 2022

Please read carefully before starting to solve the problems.

General Info

A test problem is organized as a set of related procedures (functions) that have to be implemented in order to earn points. The solution is checked by Yandex.Contest environment. It requires you to upload `solution.h` containing all the necessary *declarations* and *definitions* of the types and functions created by you.

Your solution **must not** contain any definition of the `main()` method! The `main()` method is pre-defined by us and consists of a set of test procedures testing your solution in a unit-test-like manner. Some of these tests will be available for you in order to provide some hints what and how we are going to test.

Each subtask in the Contest checks only some of the methods. We explicitly state which methods must be provided in your solution and what their interface is. We give you an exact prototype for each method and you must follow it, otherwise your solution will not be compatible with the testbed and will not compile.

We design the tasks (and, hence, the solutions) so that all functions be in interaction between each other (in order to provide a proper decomposition). For each task you are expected to solve it by adding a new mentioned function and manage its interaction with others. In your turn, you may expect that if you submit a solution that properly works at your side (with the mentioned restrictions, obviously; say, w/ no `main()` function in `solution.h`), it will also not conflict with the stubs needed to call your code. However, *just in case*, be ready to adjust the submitted solution only to the mentioned functions, commenting out the remaining code.

Tasks

There is a set of records about dishes, stored in a text file. Each dish is represented by a single line consisting of space-separated tokens, as follows:

```
Abnabat 4 Carrot Nuts Water Onion
```

Here the first word refers to a dish itself. Then it is followed by a number *k* indicating a count of ingredients. Finally, *k* words follow representing ingredients of the dish.

Programmatically a dish is represented by a custom structure `Dish` with two fields, namely dish name and dish ingredients:

```
struct Dish {
    std::string name;
    std::set<std::string> ingredients;
};
```

Each ingredient has a calorie content represented in the file by a single line consisting of space-separated tokens, as follows:

```
Potato 142
```

Programmatically a collection of ingredients is represented by a map as follows:

```
using Ingredients = std::map<std::string, int>;
```

A collection of dishes is represented by a vector object:

```
using Dishes = std::vector<Dish>;
```

You don't modify the structure. If you need to store something in addition for solving the tasks below, consider any separate storage.

Input file (streamed) description

All methods reading input data deal with a single sequential stream internally associated with a text file. While testing, you only need to provide the correct path to the data file in `main.cpp`, and work with the given stream after that.

The whole input data are structured as follows:

- the first line contains a number *m* that is the number of ingredient descriptions;
- the following *m* lines representing *m* ingredient descriptions;
- next line contains a number *n* that is the number of dishes;
- the following *n* lines representing *n* dishes.

A text file `data/dishes.txt` is an example of given data.

General notes

- While implementing functions, consider a proper decomposition.
- If you include any header file, leave the corresponding `#include` directives while submitting a solution in Yandex.Contest.

Task 1: Read ingredients data from a stream [up to 0.2]

Implement a function `readIngredients()` loading data about ingredients from a given stream associated with a text file. The function returns a collection of ingredients as an `Ingredients` collection.

All the needed details are provided in the skeleton code of `solution.h`. There is a predefined skeleton for the function. You are not allowed to modify its header, however you are free to provide entirely your own implementation and, hence, remove any pre-defined code inside the body.

Note: There is a similar method `readDishes` reading dishes from a given stream and returning them as a collection `Dishes`. You don't need to re-implement it, so simply consider it as an example and use it in tests.

What is checked for this task?

Only `readIngredients()` function. See `test1()` in `main()` for example.

What to submit to Yandex.Contest?

A `solution.h` file containing a correct definition of this function and all decomposition functions and all necessary header inclusions. There shouldn't be `main()` inside.

Task 2: overload `operator<<` for the `Dish` class [up to 0.2]

Overload `operator<<` for `std::ostream` and the `Dish` custom structure. The operator outputs a dish description as in the following example:

```
Abnabat: Carrot, Nuts, Onion, Water
```

Mention the following points: there is only one space after `:` and `,` characters; there is no comma in the line's end; there is no new line wrapping `\n`.

What is checked for this task?

An ability to apply the following semantics:

```
Dishes dish = ...
std::cout << dish;
```

See `test2()` in `main()`, for example.

What to submit to Yandex.Contest?

A `solution.h` file containing a correct definition of two operator functions and all decomposition functions (if needed) and all necessary header inclusions. There shouldn't be `main()` inside.

Task 3: Calculate calories for a dish [up to 0.2]

Implement a function `calcCalories()` which obtains a `Dish` object and a collection of ingredients `Ingredients` and calculates the calorie count for the given dish based on the given ingredients table.

A calorie count is calculated as a sum of calories of all ingredients for a given dish. Specific calorie values are extracted for the ingredients table.

For example, for a dish

```
Abnabat: Carrot, Nuts, Onion, Water
```

its ingredients together with calories are the folowing:

- Carrot → 27, Nuts → 140, Onion → 3, Water → 1

So, the total calories for the dish *Abnabat* is 171.

What is checked for this task?

Only `calcCalories()` function. See `test3()` in `main()` for example.

What to submit to Yandex.Contest?

A `solution.h` file containing a correct definition of this function and all decomposition functions and all necessary header inclusions. There shouldn't be `main()` inside.

Task 4: overload `operator<<` for a `DishesIngredients` pair [up to 0.2]

There is an auxiliary pair object `DishesIngredients` declared as follows:

```
using DishesIngredients = std::pair<Dishes&, Ingredients&>;
```

It ties together two collections, that is `Dishes` and `Ingredients`, *by reference*. Such pair is used to pass a list of dishes together with an ingredients table in the cases when they are used together.

Overload `operator<<` for `std::ostream` and the `DishesIngredients` pair. The operator outputs a list of dishes, one per line, each dish is followed by a calorie count. See an example:

```
Abnabat: Carrot, Nuts, Onion, Water; 171
```

Mention the following points: use `operator<<` for the Dish class to output the main part of the dish description; there is only one space after `:` character.

For example see the code of `operator<<` for the `Dishes` collection.

What is checked for this task?

An ability to apply the following semantics:

```
Ingredients ingredients = ...
Dishes dishes = ...
DishesIngredients di {dishes, ingredients};
std::cout << di;
```

See `test4()` in `main()` for example.

What to submit to Yandex.Contest?

A `solution.h` file containing a correct definition of the operator functions and all decomposition functions (if needed). There shouldn't be `main()` inside.

Task 5*: Sort dishes in a descending order of calories [up to 0.2]

This is an advanced task, for the maximum grade of 10 pts.

Implement a function `sortDishesByCalories()` which sorts a given vector of dishes via a provided `DishesIngredients` object in a **descending order** of calories.

The implementation must be done using `std::sort`. For doing that, you have to implement a custom comparator datatype `Comparator`. As usual, it requires overloading of the `operator()`.

An important point is that the comparator object must consider a calorie table for each comparison, so this table object must be stored in the comparator. For doing that, consider the `const Ingredients& ingrs` field.

What is checked for this task?

The function `sortDishesByCalories()` and the completed version of the comparator structure `Comparator`. See `test5()` in `main()` for example.

What to submit to Yandex.Contest?

A `solution.h` file containing a correct definition of this function and the comparator class. There shouldn't be `main()` inside.