CS100 Homework 7 (Spring, 2022)

Deadline: 2022-05-30 23:59:59

Late submission will open for 24 hours after the deadline, with -50% point deduction.

Problem 1. Gradesheet

In this problem, you are going to implement a simple `Gradesheet` class. The students' grades are recorded in a grade sheet. Each student has their name, student number and grade, which are covered in the `Grade` class (See the code for details). You need to implement the all the getters and setters of `Grade`.

All grades are stored in a `std::vector` inside a `Gradesheet` class. You need to implement all the provided member functions as follows.

- Default constructor and destructor. The default constructor initializes an empty grade sheet.
- Return the size of gradesheet, that is the number of records in this gradesheet.

```
std::size_t size() const;
```

- Compute the average of all students' grades.

```
double average() const;
```

- Add a new student's grade into the gradesheet. If the student's name or student number already exists, do nothing and return `false`. Otherwise, add the grade and return `true`.

```
bool addGrade(const Grade &grade);
```

- Return the grade of the student with the given name or student number. Return `-1` if such student is not found.

```
double findByName(const std::string &name);
double findByNumber(int number);
```

- Overloading subscript operator. Returns a reference to the `i`-th `Grade` recorded in the grade sheet. The numbering starts with 0.

```
Grade& operator[](size_t i);
const Grade& operator[](size_t i) const;
```

```
/------------------------------\
|Name      Number    Grade     |
|------------------------------|
|Alex      1         100       |
|Alice     23        90        |
\------------------------------/
```

- Overloading `operator<<` which applied to an output stream. The desired output should be like:

  There are 3 columns in total. Each column is 10 characters wide. There are 30 '-' for those dashed lines. Each row contains the student name, number and grade. Notice that all the numbers and strings should be **left-aligned**. For the grades, you should print the number with a precision of 3 (use `std::precision(3)`). DO NOT forget to add the boundary as shown in the example above.

  ```
  friend std::ostream& operator<<(std::ostream& os, const Gradesheet& sheet);
  ```

- Sort the gradesheet by student name in **alphabetical** order or by student number in **ascending** order or by grade in **descending** order. You can use `std::sort` from `<algorithm>` for simplicity. We also provide you 3 classes (`NameComparator`, `NumberComparator`, `GradeComparator`). You should overload the `operator()` of each class for comparison.

  ```
  void sortByName();
  void sortByNumber();
  void sortByGrade();
  ```

Sample code

```
Gradesheet sheet;
sheet.addGrade(Grade("Bob", 1, 95));
sheet.addGrade(Grade("Carl", 2, 100));
sheet.addGrade(Grade("Alex", 3, 90));
sheet.sortByGrade();
std::cout << "size == " << sheet.size() << "\n" << sheet;
```

The output is

```
size == 3
/------------------------------\
|Name      Number    Grade     |
|------------------------------|
|Carl      2         100       |
|Bob       1         95        |
|Alex      3         90        |
\------------------------------/
```

Notes

- The `Grade` class is very simple with only one constructor, some getters and some setters. The constructor initializes every member with the corresponding parameter. Complete this class first.

- You may find `std::setw`, `std::setprecision` and `std::left` defined in standard library file `<iomanip>` helpful to implement the `operator<<`.
- It is guaranteed that the length of the student name and number will not exceed 10.

Submission Guideline

When you submit your code, your main function will be replaced by one on OJ. You MUST NOT modify the definition of the class. Otherwise, you will NOT receive any scores.

Problem 2. Singly Linked-List

In this task, you will need to write an STL-style templated singly-linked-list and its iterator. Considering that many of you may have only a vague idea of template programming, we have provided a framework for you so that you don't have to learn too many things. Moreover, since the compiler **won't generate any code** for templated functions that are not used, we also provide a simple test which involves some compile-time checks and some runtime tests. You can paste or `#include` your code at the beginning of `7-2_simple_tests.cpp` to run the simple test.

In the framework, first we have the `Slist_node<T>` class. This is a simple structure that defines the node of the linked-list. **You probably need to add some constructors for this class.**

The Iterator

Then it comes the `Slist_iterator<T, is_const>` class, which defines the iterator of the linked-list. `T` is the type of values that can be obtained by dereferencing the iterator. `is_const` is a bool value denoting whether this iterator is a `const_iterator`. A `const_iterator` differs from a regular `iterator` in that the value that is being pointed to cannot be modified through a `const_iterator`. In this sense, dereferencing a `const_iterator` should return a reference-to-`const`, and on a `const Slist<T>` the `begin()` and `end()` functions should return `const_iterator`s.

Every STL-style iterator should have the following type aliases: `value_type`, `difference_type`, `pointer`, `reference` and `iterator_category`. We have defined them for you and have provided explanation in the framework. The `m_cur` member points to the node containing the element that the iterator is pointing to.

The iterator should meet the requirements of a 'forward-iterator': It must support `operator*` (dereference operator), `operator->` (arrow member-access operator) and `operator++` (both prefix and postfix). The `operator->` might be a little bit tricky, so you only need to implement the other three operators. We have provided the declarations of these functions and please **DO NOT modify them**, or you may encounter compile-error.

Your operators must behave in consistency with the built-in behaviors. For example, `++iter` returns reference to the object '`iter`', while `iter++` returns a copy of the object before incrementation.

The Slist

Now let's begin implementing the core part of this task. As in `Slist_iterator`, there are some type alias members that every STL container must have, and we have provided them for you. The `Slist<T>` has two data members. `m_head` points to the head of linked list, and `m_length` is the number of elements stored in the list.

First, the `Slist` needs a default-constructor, a copy-constructor, a copy-assignment operator and a destructor. You need to define them on your own. The requirements of these functions are stated in the framework. In particular, **we highly recommend using the 'copy-and-swap' technique** to implement the copy-assignment operator, which saves you a lot of work. If you don't know what it is, you can refer to the reference solution of hw5-1.

Then you need to implement some operations on the linked-list. These operations are `push_front`, `pop_front`, `insert_after` and `erase_after`. The requirements are stated in the framework. We have provided the declarations for you and please **DO NOT modify them**, or you may encounter compile-error. Note that we have defined the `base()` and `next()` functions in the `Slist_iterator` class, which you may find helpful.

The `Slist` should also support `size()`, `empty()` and `clear()`, which behave the same as in every STL container. `size()` should return a value of the type '`size_type`', denoting the number of elements stored in the list. `empty()` returns `true` if and only if the list contains no elements. `clear()` removes all the elements in the list.

The well-known '`begin()`' and '`end()`' functions have been implemented for you.

In the end, you will also need to implement the `operator==` and `operator<` of `Slist`. Two `Slist`s are thought of as equal if and only if they are of the same length, and every pair of corresponding elements are equal. The `operator<` compares two `Slist`s in the lexicographical order. **We strongly suggest** using `std::equal` and `std::lexicographical_compare` defined in `<algorithm>`, which will save you a lot of work.

Please do not declare unnecessary `friend`s.

Special Requirements

When it comes to generic programming, it is best practice to minimize the number of requirements placed on the unknown types.

- The template argument `T`, which denotes the type of the elements stored in the linked-list, **may not be default-constructible or copy-assignable**. Considering that you have not learned about *variadic templates*, *perfect forwarding* and *move semantics*, it is guaranteed that `T` is copy-constructible. We have provided a `Special_type` in the simple tests which is neither default-constructible nor copy-assignable. Make sure that your `Slist` works well when `T = Special_type`.
- It is guaranteed that `operator==` will be called only when

  ```
  bool operator==(const T &, const T &)
  ```

  is defined. `operator<` will be called only when

  ```
  bool operator<(const T &, const T &)
  ```

  is defined. Note that your `operator<` should not depend on other relational operators of `T`, like `operator>` or `operator!=`. The `Special_type` provided in the simple tests is an example, on which only `operator==` and `operator<` are defined.

Notes

- Do not modify any code we have written for you, or you may encounter compile-error.
- Remove the comments when they are not needed, in order to improve readability of your code.

Submission Guideline

Submit your code to the OJ. It should contain the `Slist_node`, `Slist_iterator` and `Slist`, as well as some non-member operators. Do not contain any tests in your submission.