CS100 Homework 6 (Spring, 2022)

Deadline: 2022-05-15 23:59:59

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

Problem 1. Matrix

In this problem, you will implement a Matrix class. A **Matrix** class contains the following member functions:

- Constructor: Create a matrix of size `rows`×`cols` and set all the elements to 0.

```
explicit Matrix(Index rows, Index cols);
```

- Copy constructor: Copy all elements from the other `Matrix` instance.

```
Matrix(const Matrix &other);
```

- Copy-assignment operator: **You should make sure that the behavior is correct when self-assignment happens**.

```
Matrix& operator=(const Matrix &other);
```

- Destructor: Make sure all the dynamically allocated memory is deallocated.

```
~Matrix();
```

- Return the number of rows and columns of the matrix.

```
Index rows() const;
Index cols() const;
```

- Return the element at position `(r,c)`. Note that the indices start from zero.

```
Scalar& at(Index r, Index c);
const Scalar& at(Index r, Index c) const;
```

- Operators: Matrix addition, subtraction and multiplication.

```
Matrix operator+(const Matrix &other) const;
Matrix operator-(const Matrix &other) const;
Matrix operator*(const Matrix &other) const;
```

- Return the transpose of the matrix

```
Matrix transpose() const;
```

- Return a submatrix of size `(p,q)` starting at `(r,c)`

```
Matrix block(Index r, Index c, Index p, Index q) const;
```

- Return the trace of the matrix.

```
Scalar trace() const;
```

- Return the determinant of the matrix

```
Scalar determinant() const;
```

Notes

- You need to implement all the provided member functions of the `Matrix` class. You do not need to handle the input or output.
- You can design your own way to store the data. We encourage you to use STL containers such as `std::vector`.
- You should use `new/new[]` and `delete/delete[]` for memory allocation. **Using `malloc` and `free` will cause trouble in C++, unless you really understand what you are doing**. You need to make sure that no **memory leak** occurs in your implementation.
- All the testcases are valid operations. You don't need to worry about misuse. For example,
  - `trace` and `determinant` will only be used for square matrix.
  - All indices and matrix operations are valid.

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. Alarm Clock

In this problem, you are going to write C++ classes to implement an alarm clock that can add and trigger multiple alarms. Specifically, you will need to implement the `class AlarmClock` and several classes derived from `class Alarm`.

Your `AlarmClock` should keep track of its time. When created, its time will be at 00:00.

Your `AlarmClock` should store all the alarms with an `std::vector<Alarm*>`, an `std::vector` of pointers to `Alarm`.

We will simulate 3 days on your alarm clock. For every day, you can add alarms by input when the day begins (00:00). Then, a member function `TimeElapse()` is called on your clock for 24×60 times, simulating every minute in the day. If at some moment, your clock has alarms set on that time, these alarms will "go off" by printing a message.

Specifically, there are three member functions of `class AlarmClock` that you must implement:

- `void AlarmClock::AddAlarm(Alarm* alarm);`

  This function adds to your alarm clock a new alarm of type `Alarm*`. The parameter `alarm` should be created by the C++ keyword `new`.

- `void AlarmClock::Input();`

  This function is called at the beginning of each day. The user may input to add alarms to your clock. See the input section below for details.

- `void AlarmClock::TimeElapse();`

  This function first goes through all alarms on the clock, triggering any that is set **at the alarm clock's current time**, and **then proceeds the alarm clock's time by one minute**. (In other words, one minute elapses on this clock.) If any alarm set at the current time is found, call `Alarm::Trigger()` on it. When there are multiple alarms found, they should trigger in the order they are added.

There are two kinds of alarms that you can set on your clock. One is called a **repeatable alarm**, which will trigger every day at the same time; the other is called a **snooze-able alarm**, which does not repeat, but has a "snooze" button. If you choose to snooze, the alarm will go off again **10 minutes later**. You can also snooze again, or for as many times as you want. Repeatable alarms do not have the snooze feature. Therefore, these two different types of alarms must inherit from an abstract base class Alarm.

The common traits of the two alarms should be stored in their parent class `Alarm`. An `Alarm` should store information about when it will go off. Also, an `Alarm` has a name that can be stored in an `std::string`.

The class Alarm has a pure virtual function `virtual void Trigger() = 0;`, which the two child classes, RepeatableAlarm and SnoozeableAlarm should implement in different ways:

- When a `RepeatableAlarm` triggers, it should print a line in the form below:

  `Alarm <NAME> has triggered at <TIME>.`

  `<TIME>` is a 24-hour time. For example:

  `Alarm CS100 has triggered at 10:00.`

  Remember, repeatable alarms will trigger every day at their set time.

- When a `SnoozeableAlarm` triggers, it should print a line in the form below:

  `Alarm <NAME> has triggered at <TIME>. Snooze? (Y/N)`

  `<TIME>` is a 24-hour time. For example:

  `Alarm MyWeekend has triggered at 08:00. Snooze? (Y/N)`

  It then waits for user input. If the input is a "Y", this snooze-able alarm should modify itself, setting its time to 10 minutes later (08:10 in this case), and will also be snooze-able then. If the input is an "N", this alarm will be "turned off", being still on the alarm clock but will never trigger, even at the same time of the next day. Any input other than "Y" are considered as an "N".

Input description

The program simulates 3 days on your alarm clock.
At the beginning of each day, new alarms will be added in the following format:

- The first line contains a number `N`, indicating the number of alarms to be added.
- Each of the next `N` lines indicate an alarm, in the following format:

  `<TYPE> HH:MM <NAME>`

  `<TYPE>` is a single character, either `'R'` or `'S'`, indicating a [R]epeatable alarm or a [S]nooze-able alarm.

After these alarms are added, the main function calls `TimeElapse()` for 24×60 times. If any snooze-able alarm goes off during this process, there will be an additional line of input, containing either "Y" or "N".

Output description

Apart from output produced in the given main function, for each triggered alarm, there will be a line of output.

Sample input (red) and output (black):

```
Do you want to add any alarms?
2
R 10:00 CS100
S 08:00 MyWeekend
```

```
Alarm MyWeekend has triggered at 08:00. Snooze? (Y/N)
Y
Alarm MyWeekend has triggered at 08:10. Snooze? (Y/N)
Y
Alarm MyWeekend has triggered at 08:20. Snooze? (Y/N)
N
Alarm CS100 has triggered at 10:00.
A day has elapsed.
Do you want to add any alarms?
1
R 15:00 afternoon
Alarm CS100 has triggered at 10:00.
Alarm afternoon has triggered at 15:00.
A day has elapsed.
Do you want to add any alarms?
0
Alarm CS100 has triggered at 10:00.
Alarm afternoon has triggered at 15:00.
A day has elapsed.
```

Problem 3. Reference Counting

Suppose you are the designer of a class `Point`, which represents a point on the 2-d coordinate system.

```
class Point {
  double x{0}, y{0};
  std::string label;

 public:
  Point(double x0, double y0, const std::string &l = "")
      : x(x0), y(y0), label(l) {}
  Point() = default;
  double get_x() const {
    return x;
  }
  double get_y() const {
    return y;
  }
  std::string get_label() const {
    return label;
  }
  Point &set_x(double x0) {
    x = x0;
    return *this;
  }
  Point &set_y(double y0) {
    y = y0;
    return *this;
  }
  Point &set_label(const std::string &l) {
    label = l;
    return *this;
  }
};
```

The interfaces of this class might be used like this:

```
int main() {
  Point p(3, 4, "A");
  std::cout << p.get_label()
            << "(" << p.get_x() << ", " << p.get_y() << ")"
            << std::endl;
  p.set_x(5).set_y(6);
  std::cout << p.get_label()
            << "(" << p.get_x() << ", " << p.get_y() << ")"
            << std::endl;
  return 0;
}
```

The output would be

```
A(3, 4)
A(5, 6)
```

Suppose the user wants to create some `Point` objects dynamically. Needless to say, dealing with dynamic memory in C/C++ is error-prone. Even experienced programmers may forget to deallocate the memory when it is not used, or may dereference an invalid pointer. For example, the following code

```
Point *p1 = new Point(3, 4, "A"), *p2 = new Point(5, 6, "B");
p1 = p2;
```

has **memory leak**, because after the assignment no pointer will be pointing to the object that `p1` pointed to any more. As a result, there's no way to destroy that object or deallocate that block of memory.

It seems that we should `delete` it before the assignment:

```
Point *p1 = new Point(3, 4, "A"), *p2 = new Point(5, 6, "B");
delete p1;
p1 = p2;
```

Now the problem of memory leak is fixed. However, things are not as simple as that. In practice, your program may look like this:

```
Point *p1 = some_value(), *p2 = some_other_value();
f(p1, p2);
g(p1);
h(p2);
Point *p3 = some_value_related_to(p1);
// ...
// ...
// Should we delete p1 now?
p1 = p2;
```

It becomes much more difficult to determine whether we should `delete p1` right before the last assignment! In essence, the problem boils down to **knowing how many pointers are pointing to an object** at a particular moment.

To manage the memory correctly, we will design a 'smart pointer' that can, somehow, be aware of how many **smart pointers** are sharing the same object. Such 'smart pointer' is called a **reference-counted handle**. First, we need to attach a counter to a `Point` object by defining a helper class:

```
class Point_counted {
  friend class Point_handle;

  // All the members are private.
  Point p;
  unsigned cnt; // The reference counter.

  // You may define some helper member functions if you need.
};
```

Then we can define our handle class, named `Point_handle`.

```
class Point_handle {
  Point_counted *ptr;
};
```

The handle class contains a pointer to the `Point_counted` type, and should behave as follows:

- When an object is created by the handle, the reference counter is set to `1`.
- When `ph1` is copy-initialized from `ph2`, i.e.

  ```
  Point_handle ph1 = ph2;
  ```

  `ph1.ptr` and `ph2.ptr` should point to the same object, with its reference counter increased by `1`.
- When `ph1` is copy-assigned from `ph2`, i.e.

  ```
  Point_handle ph1 = some_value(), ph2 = some_other_value();
  ph1 = ph2;
  ```

`ph1.ptr` and `ph2.ptr` should point to the same object, with its reference counter increased by `1`. The reference counter of the object that `ph1.ptr` pointed to before the assignment should be decreased by `1`.

- When a handle is about to be destroyed, the reference counter of the object that it points to should be decreased by `1`.
- Whenever the reference counter of an object is decreased to `0`, that object should be `delete`d.

Moreover, the handle class should provide some interfaces:

- Constructors which create a `Point` object from the given arguments, set its reference counter to `1`, and let `ptr` point to it.

  ```
  Point_handle::Point_handle();
  Point_handle::Point_handle(const Point &);
  Point_handle::Point_handle(double, double, const std::string & = "");
  ```

  Note that the default constructor should **create a `Point` object by default**.

- Member function `ref_count` that returns the value of the reference counter of the object.
- Member functions `get_x`, `get_y` and `get_label`, which return the corresponding information of the point.
- Member functions `set_x`, `set_y` and `set_label`, which receive an argument and set the corresponding data to that argument. Note that you should allow the 'chained' modification like this:

  ```
  Point_handle ph = some_value();
  ph.set_x(4).set_y(10);
  ```

  In particular, we would like the handle to have **value semantics**, that is, the output of the following code should be `5` instead of `7`.

  ```
  Point_handle ph1(5, 6, "A"), ph2 = ph1;
  ph2.set_x(7);
  std::cout << ph1.get_x() << std::endl;
  ```

  To achieve this, we adopt the idea of **copy-on-write**: For `set_x`, `set_y` and `set_label`, we should ensure that the particular object that we are modifying is not used by any other handle. The way to do that is to look at the reference counter. If it is `1`, then just do the modification directly; otherwise, we let `ptr` point to a copy of the original object, and do the modification on the copy. (Do not forget to decrease the reference counter of the original object when copying.) For example, the output of the following code should be '`1, 1`'.

  ```
  Point_handle ph1(5, 6, "A"), ph2 = ph1;
  ph2.set_x(7);
  std::cout << ph1.ref_count() << ", " << ph2.ref_count() << std::endl;
  ```

Notes

- This is not the only way to implement a reference-counted handle. The class `Point_counted` will be treated as your implementation details, which we will not check.
- Although we have described how to implement a reference-counter manually, **you are allowed** to use the smart pointers provided in the standard library, which you will learn about in a few weeks.
- Do not modify the definition of the `Point` class.

Submission Guideline

- Submit your `Point_handle` class together with other helper classes (e.g. the `Point_counted` class).
- Do not include the definition of the `Point` class in your submission, or you will encounter compile-error. We will place it before your code.

More on Reference-counted Handles

The technique of reference-counting is widely adopted in practice. Recall that you don't need to care about memory management in `Python`. This is because the `Python` memory manager takes care of this behind the scenes by reference-counting.

The reference-counting handle is also a perfect choice when it comes to dynamic binding. Just think about how we can define an array of objects, each of which might have different dynamic types. Using a reference-counting handle which keeps a pointer to the base class, we can tackle the copy control and memory management problems while still keep the dynamic binding properties. You may have a try on HW5-3 if you like. :)