Messages Order

In real life applications and systems, a common component is a messaging system. Thea idea is that a sender sends messages to the recipient. The messages might be sent for example over the network. However, some network protocols don't guarantee to preserve the order of sent messages while they are received by the recipient. For example, if someone sends a text messages `hello`, `hi` and `what's up`, they might be received in the order `what's up`, `hello`, `hi`. In many systems the expected behavior is to preserve the order, so the order of sent messages is the same as the order of received messages.

In this problem, the task is to implement a software layer over the top of a network protocol sending messages in arbitrary order, in such a way that the sent messages are printed by the recipient in the order they were sent.

In the template code below, there are implementations of classes `Recipient` and `Network`.

Your task is to implement classes `Message` and `MessageFactory` according to the below specification:

Class `Message` is required to store a text value of type `std::string` and provide a public getter `const string& get_text()` which is expected to return this text value. Besides that, it should implement the `<` operator that will be used in `fix_order()` method of the recipient to fix the order of received messages. Feel free to implement any other methods and class/instance variables. In particular, you can implement any additional constructors, but make sure that you provide an empty constructor, i.e. the one without arguments.

Class `MessageFactory` is required to have an empty constructor, and implement a method `Message create_message(const string& text)` that is expected to return a `Message` object storing the value of `text` argument. Feel free to implement any other methods and class/instance variables of this class.

The locked code template will act as follows. First, it creates objects `message_factory` and `recipient`. These objects are of types `MessageFactory` and `Recipient` respectively. Then, it reads messages from the standard input, and then it will use the provided `Network` class to simulate sending the messages to the `recipient`. The `Network` class randomly shuffles the passes messages and then it passes them to the `recipient` using `recipient.receive(const Message&)` method. After all messages are delivered, the recipient uses its own method `print_messages` to print all the received messages to the standard output, and just before doing that, it uses its own `fix_order` method to fix the order of retrieved messages. For that purpose, it uses `std::sort()` algorithm to sort the `std::vector` of received messages and this is the reason your `Message` class implementation has to provide the `<` operator.

Input Format

The input is read by the provided locked code template. It contains several lines of text messages in the order that they will be sent over the network.

Constraints

- There will be at most 10 lines in the input.
- Each line will contain at most 20 characters

Output Format

The output should be produced by the provided locked code template and it is produced as described in details in the statement. The expected order of printed messages is the same as the one in the input.

Sample Input 0

```
Alex
Hello Monique!
What'up?
```

Not much :(

Sample Output 0

```
Alex
Hello Monique!
What'up?
Not much :(
```