C++ Foundation, Assignment 4

You started your first week as an intern in a software company. Your supervisor just went on holidays but he left you a message. Apparently, you should work on an old C software that's a part of the company's embedded messaging protocol. Following your supervisor's note, the software is buggy and outdated. Moreover, the code has almost no comments in it, and there is no paper documentation. Your "helpful" supervisor left some `FIXME` comments in the C code that point to potential problems in the current implementation.

As a warm-up for your internship you must rewrite the program using C++. The following sketch of the new implementation was left for you.

New implementation

The C++ program should have at least the classes: `message`, `message_broker` and `message_reader`.

It should be implemented in the *new implementation* directory of the project.

Message class

This class is based on the current `message.h` and `message_api.h` functionality.

The following device types are currently supported:

- temperature sensor,
- humidity sensor,
- pressure sensor,
- heater,
- air purifier,

The commands are always in capital letters and are device-specific.

The device types should be modelled as a *scoped enumeration*:

```
enum class TargetType{ ... };
```

This enumeration should also contain a special value `UNKNWON` that's used for messages with irregular content.

The new `message` class should contain the data members:

```
long message_id;  // message id
long target_id; // unique device id
TargetType target_type;

long long timestamp;
std::string text;
```

This class should have all its data members `private` and provide at least the following `public` functions:

- a default constructor — initialises `target_type` with an `UNKNWON` value,
- a constructor that takes the values of `target_type`, `target_id` and `text` as arguments.
- functions (accessors) for obtaining the values of all the data members.
- a function (modifier) for changing `text`.
- function `is_valid` that returns `true` if a message instance is valid and `false` otherwise (a message is valid if the `target_type` is not `UNKNWON`, and `text` is not empty).
- function `as_string` that returns a `std::string` with a textual representation of a `message` object (see the current `print_message` function for how to represent a message).

The `timestamp` and `message_id` should be initialised automatically, in the same way as in the current C implementation, in all constructors.

Also provide a function similar to `device2str` that translates a `TargetType` enumeration value into a `std::string` representation. This function doesn't have to be a member of the `message` class.

Message broker class

This class is based on the current `message_broker.h` functionality. The C message broker stores messages as a linked list and exposes a very minimalistic API. The `message_broker` class should use `std::vector` for storing `messages`:

```
std::vector<message> messages;
```

This class should have all its data members `private`. The following `public` functions should be a part of this class:

- a default constructor
- a function for adding new messages to the broker:

  ```
  void message_broker::add(const message&);
  ```

- an overload of `message_broker::add` that takes the values of `target_type`, `target_id` and `text` as arguments, creates a `message` object and adds it to the broker.
- a function that returns the total number of messages currently stored in the broker.

  ```
  int message_broker::count() const;
  ```

- a function that returns the number of messages for a device with a given id stored in the broker:

  ```
  int message_broker::count(long target_id) const;
  ```

- a functions that returns a reference to the first message stored in the broker for a device with a given id:

```
message& message_broker::peek_message_for(long target_id);
```

This function should be *const-overloaded*. Assume that such a message always exists. (A user can check it using the `message_broker::count` function.)

- a function that returns the first message stored in the broker for a device with a given id **and simultaneously removes it from the broker**. It is similar in functionality to the `next_message_for` function of the C implementation.

```
message message_broker::pop_message_for(long target_id);
```

- a function that prints all the messages for a target id passed as an argument. Overload this function with a variant that prints all the messages in the broker.

```
void print(long target_id) const;
void print() const;
```

Tip: that's how an item at a given index can be removed from a `std::vector`:

```
void remove_at_index(std::vector<message>& vec, int index){
  auto to_remove = vec.begin();
  std::advance(to_remove, index);
  vec.erase(to_remove);
}
```

To return a `message` and remove it, you'll need first to make a copy of it, then remove it and finally return the copy.

Message reader class

The `message_reader` class replaces the functionality implemented in `message_list.h` It should have:

- a constructor that takes a name of the file that contains messages as an argument.
- a function that returns the next, parsed `message` read from a file (similarly to `lst_read_next`):

```
message message_reader::read_next();
```

The parsing should be as robust as possible.

- a function `has_next` that returns a `bool` value telling whether there is a next `message` that can be read using the `read_next` function. Those two functions can be then used together:

```
message_reader reader{"messages.in"};

while(reader.has_next()){
  auto msg = reader.read_next();
  std::cout << msg.as_string();
}
```

Extra functionality (if you need a challenge)

You can also implement the following additions to your programs (from the easiest to the most challenging):

1. Use a `std::list<message>` instead of `std::vector<message>` the `message_broker` class. This will require some tweaks to the implementation: `std::list` doesn't support index access.
2. Let the `message_broker::print` take an extra parameter of type `std::ostream&`, for instance:

   ```
   void message_broker::print(long target_id, std::ostream& out =
   std::cout){

      // now instead of using:
      std::cout << "...";

      //use:
      out << "...";
   }
   ```

   This parameter has a *default value* of `std::cout`. With it, your `print` function can print to any stream you pass:

   ```
   message_broker broker{};

   //...

   // prints to the default std::cout
   broker.print(123);

   // prints to a text file:
   std::ofstream ofile{"out.txt"};
   broker.print(123, ofile);
   ```

   Why is the `out` parameter taken by a reference? Why is this reference non-constant?
3. The `message_reader` class could have an additional constructor that takes an input stream parameter:

   ```
   message_reader::message_reader(std::istream& input);
   ```

   You'll need a data member of exactly the same type: `std::istream&`. Now you'll be able to read the `messages` from any input stream, for instance:

   ```
   std::ifstream input{"messages.txt"};
   message_reader reader{input};
   ```

But you'll have to be careful — since you are storing a reference to an input stream as a member of a class, you'll have to make sure that the original stream object lives at least as long as the `message_reader` object. Alternatively, you can consider reading the whole stream, line-by-line in the constructor and storing the lines in a `std::vector<std::string>` as a data member for later use.

4. Instead of using the provided functions to timestamp the `messages`, try to use the C++ time facilities. Grab the timestamp using *the high resolution clock* and store them in your `message` class using a `time_point`.
5. Try to make the functions of the `message_broker` for extracting the messages for a devices with a given target id a bit more *snappy*. These functions are supposed to work in tandem:

```
message_broker broker;

//...

while (broker.count(123) != 0){

   // extract the next message for id 123
   auto msg = broker.pop_message_for(123);
   // do something with the msg
}
```

If each time the function `pop_message_for` is called, the iteration starts at the beginning of the vector where `messages` are stored, it will be very inefficient. Can you think of a better way, without using additional data structures?

Tip: what about remembering the last device id for which this function was called and the index at which a `message` was extracted from the vector? Be sure to account for scenarios when new `messages` are added to the broker in the meantime.