PS5A

GUITAR HERO: RINGBUFFER IMPLEMENTATION WITH UNIT TESTS AND EXCEPTIONS (PART A)

In Part A, we implement the ring buffer that will hold the guitar string position data, and write test functions and exception handling.

See http://www.cs.princeton.edu/courses/archive/spr15/cos126/assignments/guitar.html for the full assignment

IMPLEMENTATION

Write a class named RingBuffer that implements the following API:

```
capacity):
void enqueue(int16_t x)     // add item x to the end
int16_t dequeue()           // delete and return item from the front
int16_t peek()              // return (but do not delete) item from the
front
------------------------------------------------------------------------
```

Your code must `#include <stdint.h>` header that defines the standard 16-bit integer type `int16_t`.

Important notes:

1. The code should be in a pair of files named `RingBuffer.cpp` and `RingBuffer.hpp`.
2. Attempts to instantiate with a capacity less than 1 should result in a `std::invalid_argument` exception, and the error message `RB constructor: capacity must be greater than zero`.
3. Attempts to enqueue to a full buffer should result in a `std::runtime_error` exception, and the error message `enqueue: can't enqueue to a full ring`.
4. Attempts to dequeue or peek from an empty buffer should result in a `std::runtime_error` exception, and an appropriate error message.

DEBUGGING AND TESTING

- You should write a `test.cpp` file that uses the Boost functions `BOOST_REQUIRE_THROW` and `BOOST_REQUIRE_NO_THROW` to verify that your code properly throws the specified exceptions when appropriate (and does not throw an exception when it shouldn't). As usual, use `BOOST_REQUIRE` to exercise all

CPPLINT

Google's style guide is here: https://google.github.io/styleguide/cppguide.html

The `cpplint.py` file can be retrieved from https://github.com/google/styleguide/tree/gh-pages/cpplint

Save the `cpplint.py` file on your machine, and then:

```
chmod +x cpplint.py
sudo mv cpplint.py /usr/local/bin
```

Now, you can style-check a file using `cpplint.py` as an executable:

```
 cpplint.py ''filename''
```

Alternately, you could run it using Python:

```
python cpplint.py ''filename''
```

USING CPPLINT

We've agreed to turn off certain warnings. At present, you may run with:

```
cpplint.py --filter=-runtime/references,-build/header_guard --
extensions=cpp,hpp
```

ADDITIONAL FILES

Produce and turn in a `Makefile` for building your class other way)

- exactly what works or doesn't work

SUBMIT YOUR WORK

You should be submitting at least five files:

1. `RingBuffer.cpp`
2. `RingBuffer.hpp`
3. `test.cpp` (this gets downloaded as `ps5a-test.cpp`, rename it to `test.cpp`)
4. `Makefile`
5. `ps5a-readme.txt`

If you create a `main.cpp` with `printf`-style tests, you may submit that as well.

Place the files in subdirectory called `ps5a`, and archive with:

```
    tar czvf ''<archive-file-name>''.tar.gz ps5a
```

Submit using the submit utility as follows:

submit schakrab ps5a ps5a

GRADING RUBRIC

Core implementation: 4

(full & correct implementation=4 pts; nearly complete=3pts; part way=2 pts;

generate `std::invalid_argument` exception on bad constructor;

don't generate exception on good constructor;

`enqueue`, `dequeue`, and `peek` work;

generate `std::runtime_error` when calling `enqueue` on full buffer;

generate `std::runtime_error` when calling `dequeue` or `peek` on empty buffer.)

`cpplint: 2`

(Your source files pass the style checks implemented in `cpplint`)

readme.txt: 4

(Readme should say something meaningful about what you accomplished:

1 point for explaining how you tested your implementation;

1 point for explaining the exceptions you implemented;

2 points for correctly explaining the time and space performance of your RB implementation)

Total: 16