PS4

Synthesizing a Plucked String Sound (part B): *StringSound implementation and SFML audio output*

Implement the Karplus-Strong guitar string simulation, and generate a stream of string samples for audio playback under keyboard control.

StringSound Implementation

Write a class named StringSound that performs the Karplus-Strong string simulation described in Part A.

API

```
class StringSound
---------------------------------------------------------------------------------
StringSound(double frequency)         // create a guitar string sound of the
                                      // given frequency using a sampling rate
                                      // of 44,100
StringSound(vector<sf::Int16> init)   // create a guitar string with
                                      // size and initial values are given by
                                      // the vector
void pluck()                          // pluck the guitar string by replacing
                                      // the buffer with random values,
                                      // representing white noise
void tic()                            // advance the simulation one time step
sf::Int16 sample()                    // return the current sample
int time()                            // return number of times tic was called
                                      // so far
```

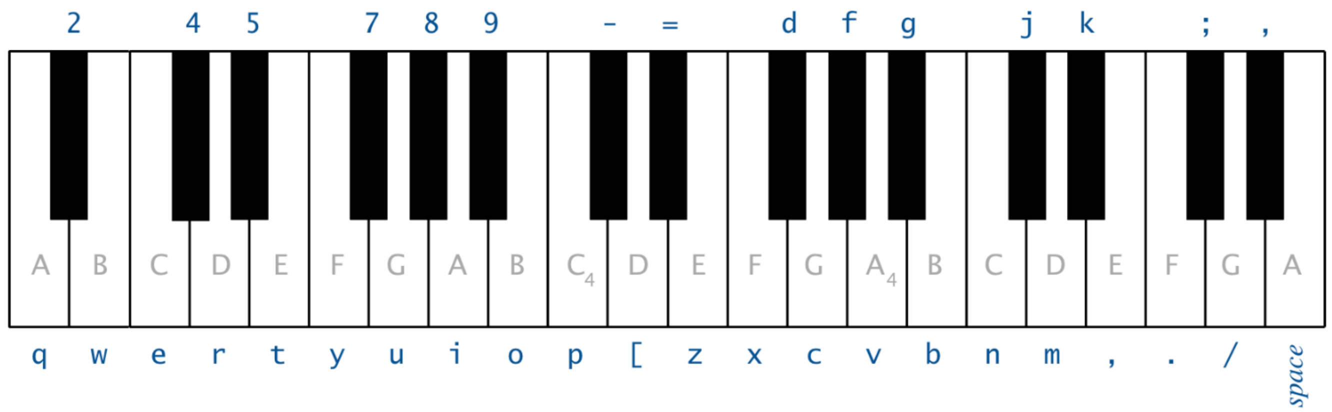
```
class StringSound {
 public:
  explicit StringSound (double frequency);
  explicit StringSound (vector<sf::Int16> init);
  StringSound (const StringSound &obj) = delete;      // no copy const
  ~ StringSound();
  void pluck();
  void tic();
  sf::Int16 sample();
  int time();
 private:
  CircularBuffer * _cb;
  int _time;
};
```

Your program `KSGuitarSim` should support a total of 37 notes on the chromatic scale from 110Hz to 880Hz. Use the following 37 keys to represent the keyboard, from lowest note to highest note:

"q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,.;/' "

This keyboard arrangement imitates a piano keyboard: The "white keys" are on the the `qwerty` and `zxcv` rows and the "black keys" on the `12345` and `asdf` rows of the keyboard (see Pic.1).

The i^{th} character of the string keyboard corresponds to a frequency of $440 \times 2^{(i - 24) / 12}$, so that the character 'q' is 110Hz, 'i' is 220Hz, 'v' is 440Hz, and ' ' is 880Hz. Don't even think of including 37 individual `StringSound` variables or a 37-way if statement!



Picture 1. Keyboard

- In the `StringSound` private member variables declarations, you must declare a pointer to a `CircularBuffer` rather than declaring a `CircularBuffer` object itself. Then in the `StringSound` constructor you must use the `new` operator.
    - This is because you can't allow the CircularBuffer to be instantiated until the `StringSound` constructor is called at run time (you don't know how big a CircularBuffer to make until given the frequency of the string).
  - See http://stackoverflow.com/questions/12927169/how-can-i-initialize-c-object-member-variables-in-the-constructor for an explanation.
    - Because the CircularBuffer contained in the guitar string class will be a pointer to a CircularBuffer, you'll need to use the dereference operator (`*`) to get at the CircularBuffer object itself.
    - Remember to explicitly `delete` the CircularBuffer object in the `StringSound` 's destructor.
- In the `StringSound(double frequency)` constructor, you must using the ceiling function when calculating the size of the CircularBuffer.
  See http://www.cplusplus.com/reference/cmath/ceil/ for details.
- In the `pluck` method, you must fill the guitar string's CircularBuffer with random numbers over the `int16_t` range. `int16_t` is a short integer, which can hold values from `-32768` to `32767`.

- Also in `pluck`, the guitar string's circular buffer might already be full. So you should either empty it (by dequeuing values until it's empty), or by deleting it and making a new one which you'll then fill up.

Or, you could add a new method to your CircularBuffer, `empty()`, which would set the `_first` and `_last` index member variables to `0`, and the `_full` boolean to `false`. (This would be the most efficient solution.)

Testing your StringSound implementation

Before you proceed to generate sound, test that your `StringSound` is implemented correctly!
Use C++ exceptions for error handling.

SFML Audio Output

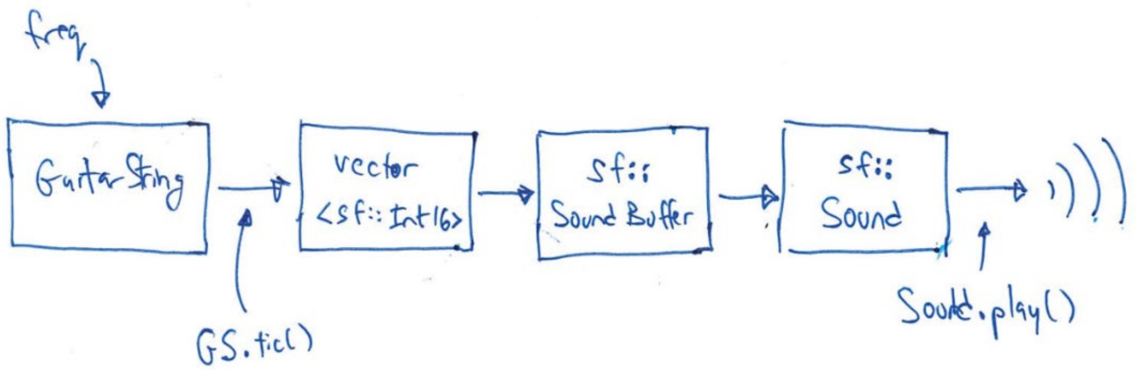
There are two parts of generating audio:
(1) getting values out of the `StringSound` object and into `SFML` audio playback object, and
(2) playing the audio objects when key press events occur.

Getting samples out of StringSound and into SFML Sound

For SFML, we have to have an existing `sf::StringSound` that's created with a vector of sound samples. This `StringSound` is created from a vector of `sf::Int16`s.

Then we create an `sf::Sound` object from the `sf::SoundBuffer`. The `sf::Sound` object can then be played.

So the whole sequence is:



Playing SFML Sounds when key presses occur

We'll use SFML to create an electronic keyboard:

- When the "a" key is pressed, a sound corresponding to concert `A` (440 Hz) should be played.
- When the "c" key is pressed, a `C` note should be played.

To handle the keypress events, we'll open an SFML window, and look for `sf::Event::KeyPressed` events.

When we get one, we'll see if its `event.key.code` is equal to `sf::Keyboard::A` or `sf::Keyboard::C`.

If so, we'll play the appropriate sound.

See the `SSLite.cpp` demo file for how to do this. `SSLite.cpp` is sample code that when given a correct implementation of `StringSound`, will play a 440 Hz A string when the “a” key is pressed, and the corresponding C note when the “c” key is pressed.

In the first half of the code, two `StringSound` objects are created (one for each frequency), and each is cranked to produce a stream of audio samples that are loaded into a `sf::Int16` vector. Those vectors are made into `sf::SoundBuffer`s, and those are made into playable `sf::Sound` objects.

In the second half of the code, an `SFML` window and event loop is set up to play the sounds when the “a” or “c” keys are pressed.

Implementation

For our implementation, we actually need three parallel arrays (please use vectors):

- a vector of 37 `sf::Int16` vectors. Each individual `sf::Int16` vector holds the audio sample stream generated by one `StringSound`.
- a vector of 37 `sf::SoundBuffer`s. Each `SoundBuffer` object contains a vector of audio samples.
- a vector of 37 `sf::Sound`s. Each `Sound` object contains a `SoundBuffer`. (It's the `Sound` object that can finally be played.)

You don't need a vector of `StringSound`s. Once you've `pluck`ed it and `tic`ed it a bunch of times to get the sound samples out of it—and stored into the `Int16` vector—you can throw it away and make a new one for the next frequency.

Extra credit

For extra credit, make a version of the program that makes a different sound. Modify the algorithm to get a sound that resembles drum, chirp, piano, or anything other than the guitar.

This sound doesn't have to simulate a specific instrument. Here's a couple of ideas:

1. Make your algorithm vary the number of samples on the queue as the sound is being synthesized, producing a frequency chirp. For example, for each 100 times that `tic()` is called, remove 100 samples from the queue, but only re-insert 99 samples. This will produce an up-frequency chirp (make sure to stop removing samples when the queue is almost empty, so that `peek()` and `dequeue()` don't throw exceptions for empty queue.)
2. Change the low-pass filter so it leaves some of the noise in the buffer for longer, resulting in a "noisier" sound - this will sound more like a percussion instrument. One way to do this is to mix 90% of the last sample and 10% of the second-last sample (guitar sound uses 50%/50% mix.)

Submit your work

You should be submitting at least five files:

- Your `CircularBuffer.cpp` and associated `CircularBuffer.h`
- Your `StringSound.cpp` and its `StringSound.h`
- Your `KSGuitarSim.cpp` file
- A `Makefile` that builds an executable named `KSGuitarSim`.
- A filled-in copy of the `ps4b-readme.txt`

Submit the archive on Blackboard.

Grading rubric

Feature	**Value**	**Comment**
`StringSound` implementation	4	full & correct implementation = 4 pts with appropriate implementation of keys (no switch statement); nearly complete = 3pts; part way=2 pts; started=1 pt -1 point for using switch/if-else statement for keys
`StringSound` C++ exceptions tests	2	Should contain C++ exceptions
`KSGuitarSim` implementation	4	transforming the `SSLite` version into the full 37-note player per assignment
	1	Use of a `lambda` expression
	1	Use of smart pointers
	2	Your code should pass cpplint
`readme`	2	Readme should say something meaningful about what you accomplished 1 point for explaining how you tested your implementation by using exceptions
`Makefile`	1	
Total	**17**	
extra credit	2	Make a version of the program that makes a different sound. Modify the algorithm to get drum, chirp, piano, or anything other than the guitar. Mention what you did in your readme.