```
CC = g++
CFLAGS = -Wall -Werror -pedantic --std=c++14
LIBS = -lboost_unit_test_framework
DEPS = CircularBuffer.h StringSound.h
SFMLFLAGS = -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio

%.o: %.cpp $(DEPS)
	$(CC) $(CFLAGS) -c $<

all: KSGuitarSim

KSGuitarSim: KSGuitarSim.o CircularBuffer.o StringSound.o
	$(CC) -g $(CFLAGS) -o KSGuitarSim $^  $(LIBS) $(SFMLFLAGS)
	cpplint --filter=-runtime/references *.cpp *.h

clean:
	rm *.o KSGuitarSim
```

```
/*
  Copyright 2015 Fred Martin,
  Y. Rykalova, 2020
  J. Daly 2022

  Edited by Anson Cheang 2022
  essentially allows the user to play
  a unique sound from 37 different keys.
  the function automatically sets up the sounds
*/

#include "CircularBuffer.h"
#include "StringSound.h"

#include <math.h>
#include <limits.h>

#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>
#include <vector>

#include <SFML/Graphics.hpp>
#include <SFML/System.hpp>
#include <SFML/Audio.hpp>
#include <SFML/Window.hpp>

#define CONCERT_A 220.0
#define SAMPLES_PER_SEC 44100

// using namespace std;

std::vector<sf::Int16> makeSamples(StringSound& gs) {
    std::vector<sf::Int16> samples;

    gs.pluck();
    int duration = 8;  // seconds
    int i;
    for (i= 0; i < SAMPLES_PER_SEC * duration; i++) {
        gs.tic();
        samples.push_back(gs.sample());
    }

    return samples;
}

int main() {
    sf::RenderWindow window(sf::VideoMode(300, 200),
     "SFML Plucked String Sound Lite");
    sf::Event event;
    char c;
    std::vector<std::unique_ptr<sf::Sound> > kSounds;
    std::vector<sf::Int16> samples;
    std::vector<std::vector<sf::Int16>> KSample;
    std::vector<std::unique_ptr<sf::SoundBuffer> > kBuffer;
    sf::Sound sound;
    sf::SoundBuffer buffer;
    std::string keys = "q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,.;/' ";

    auto func = [=] (int i) {
        std::vector<sf::Int16> samples;
        const double freq = 440.0 * pow(2.0, (i-24.0)/12.0);
        StringSound gs1(freq);
        samples = makeSamples(gs1);
```

```cpp
        return samples;
    };

    for (int i = 0; i < 37; i++) {
        KSample.push_back(func(i));
        // samples = KSample[i];
        // std::cout << samples.size() << std::endl;
    }

    for (size_t i = 0; i < KSample.size(); i++) {
        if (!buffer.loadFromSamples(&(KSample[i].at(i)), KSample[i].size(),
        2, SAMPLES_PER_SEC))
            throw std::runtime_error(
                "sf::SoundBuffer: failed to load from samples.");
        kBuffer.push_back(std::make_unique<sf::SoundBuffer>(buffer));
    }

    for (size_t i = 0; i < kBuffer.size(); i++) {
        sound.setBuffer(*kBuffer[i]);
        kSounds.push_back(std::make_unique<sf::Sound>(sound));
    }

    /* freq = CONCERT_A * pow(2, 3.0/12.0);
    StringSound gs2(freq);
    sf::Sound sound2;
    sf::SoundBuffer buf2;
    samples = makeSamples(gs2);
    std::cout << samples.size() << std::endl;
    if (!buf2.loadFromSamples(&samples[0], samples.size(), 2, SAMPLES_PER_SEC))
        throw std::runtime_error(
            "sf::SoundBuffer: failed to load from samples.");
    sound2.setBuffer(buf2); */
    int j = 0;
    while (window.isOpen()) {
        while (window.pollEvent(event)) {
            switch (event.type) {
            case sf::Event::Closed:
                window.close();
                break;

            case sf::Event::TextEntered:
                /*switch (event.key.code) {
                case sf::Keyboard::A:
                    // sound1.play();
                    break;
                case sf::Keyboard::C:
                    // sound2.play();
                    break;
                default:
                    break;
                }*/
                c = static_cast<char>(event.text.unicode);
                while (j < static_cast<int>(keys.size()) && c != keys[j]) {
                    j++;
                }
                if (j == static_cast<int>(keys.size())) {
                    throw std::runtime_error("wrong keys");
                }
                kSounds[j]->play();
                j = 0;
                break;

            default:
                break;
            }

            window.clear();
            window.display();
        }
    }
    return 0;
}
```

```
// Copyright 2022 Anson Cheang
#ifndef _HOME_IIFORCE_BADNAME_COMP4_PS4B_STRINGSOUND_H_
#define _HOME_IIFORCE_BADNAME_COMP4_PS4B_STRINGSOUND_H_

#include "CircularBuffer.h"
#include <stdint.h>
#include <iostream>
#include <cstdlib>
#include <vector>

#include <SFML/Graphics.hpp>
#include <SFML/System.hpp>
#include <SFML/Audio.hpp>
#include <SFML/Window.hpp>

class StringSound {
 public:
explicit StringSound(double frequency);
explicit StringSound(std::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;
};

#endif  // _HOME_IIFORCE_BADNAME_COMP4_PS4B_STRINGSOUND_H_
```

```cpp
// Copyright 2022 Anson Cheang

// This class is used to create the sound used by KSGuitarSim.cpp

#include "StringSound.h"
#include <time.h>
#include <iostream>
#include <cmath>
#include <vector>
#include <iterator>
#include <random>
#include <chrono> //NOLINT

#define SAMPLING_RATE 44100

StringSound::StringSound(double frequency) {
    if (frequency <= 0) {
        throw std::invalid_argument(
            "StringSound(double frequency) : cant accept values of 0 or l
ower");
    }
    size_t _size = ceil(SAMPLING_RATE/frequency);
    _cb = new CircularBuffer(_size);
    _time = 0;
}

StringSound::StringSound(std::vector <sf::Int16> init) {
    _cb = new CircularBuffer(init.size());
    std::vector<sf::Int16>::iterator p;
    p = init.begin();
    while (p != init.end()) {
        _cb->enqueue(*p);
        p++;
    }
    _time = 0;
}

StringSound::~StringSound() {
    delete _cb;
    _cb = nullptr;
}

void StringSound::pluck() {
    if (_cb->size() <= 0) {
        while (!_cb->isEmpty()) {
            _cb->dequeue();
        }
    }

    unsigned int secret = std::chrono::system_clock::now().time_since_epo
ch().count();//NOLINT
    std::mt19937 gen(secret);
    while (!_cb->isFull()) {
      // generate random number
      std::uniform_int_distribution<int16_t> dist(-32768, 32767);
      _cb->enqueue(dist(gen));
      }
}

void StringSound::tic() {
    if (!(_cb->isFull())) {
        throw std::runtime_error("Tic() : The list isn't full");
    }
    // size_t size = _cb->size();
    sf::Int16 val1 = _cb->dequeue();
    sf::Int16 val2 = _cb->peek();
    sf::Int16 new_val = .996 * ((val1 + val2) / 2);

    _cb->enqueue(new_val);

    _time++;
}

sf::Int16 StringSound::sample() {
    if (_cb->isEmpty()) {
        throw std::runtime_error("sample() : _cb is empty");
    }
    sf::Int16 sample = _cb->peek();

    return sample;
}

int StringSound::time() {
    return _time;
}
```

```
// Copyright 2022 Anson Cheang
#ifndef _HOME_IIFORCE_BADNAME_COMP4_PS4B_CIRCULARBUFFER_H_
#define _HOME_IIFORCE_BADNAME_COMP4_PS4B_CIRCULARBUFFER_H_

#include <stdint.h>
#include <cstdlib>
#include <deque>

class CircularBuffer {
 public:
CircularBuffer(size_t capacity);  // create an empty ring buffer,
// with given max capacity
size_t size();  // return number of items currently in the buffer
bool isEmpty();  // is the buffer empty (size equals zero)?
bool isFull();  // is the buffer full (size equals 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
unsigned int getCap();

 private:
std::deque<int16_t> list;
size_t currentSize;
unsigned int maxCapacity;
};

#endif  // _HOME_IIFORCE_BADNAME_COMP4_PS4B_CIRCULARBUFFER_H_
```

```
// Copyright 2022 Anson Cheang

// This is used to make the storage for the buffer.

#include "CircularBuffer.h"
#include <iostream>
#include <string>

CircularBuffer::CircularBuffer(size_t capacity) {
    std::string message =
    "CircularBuffer constructor: capacity must be greater than zero";
    if (capacity < 1) {
        throw std::invalid_argument(message);
    }
    currentSize = 0;
    maxCapacity = capacity;
}

size_t CircularBuffer::size() {
    return list.size();
}

bool CircularBuffer::isEmpty() {
    return list.size() <= 0;
}

bool CircularBuffer::isFull() {
    return list.size() == maxCapacity;
}

void CircularBuffer::enqueue(int16_t x) {
    if (isFull()) {
        throw std::runtime_error("enqueue: can't enqueue to a full ring size");
    }
    list.push_back(x);
}

int16_t CircularBuffer::dequeue() {
    if (isEmpty()) {
        throw std::runtime_error("dequeue: can't dequeue an empty ring");
    }
    int16_t val = list.front();
    list.pop_front();
    return val;
}

int16_t CircularBuffer::peek() {
    if (isEmpty()) {
        throw std::runtime_error("peek: can't peek an empty ring");
    }
    return list.front();
}

unsigned int CircularBuffer::getCap() {
    return maxCapacity;
}
```

```
// Copyright 2022 Anson Cheang
// test.cpp
// updated 1/30/22-1/31/22
// updated by Anson Cheang

#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Main

#include "CircularBuffer.h"
#include "StringSound.h"

#include <iostream>
#include <sstream>
#include <string>
#include <boost/test/unit_test.hpp>

BOOST_AUTO_TEST_CASE(sixteenBitsThreeTaps) {
    BOOST_REQUIRE_THROW(StringSound l(0), std::invalid_argument);
    BOOST_REQUIRE_THROW(StringSound l2(-1), std::invalid_argument);
    BOOST_REQUIRE_NO_THROW(StringSound l3(5));
    StringSound l3(5);
    BOOST_REQUIRE_THROW(l3.tic(), std::runtime_error);
    BOOST_REQUIRE_THROW(l3.sample(), std::runtime_error);
    l3.pluck();
    BOOST_REQUIRE_NO_THROW(l3.tic());
    BOOST_REQUIRE_NO_THROW(l3.sample());
}
```