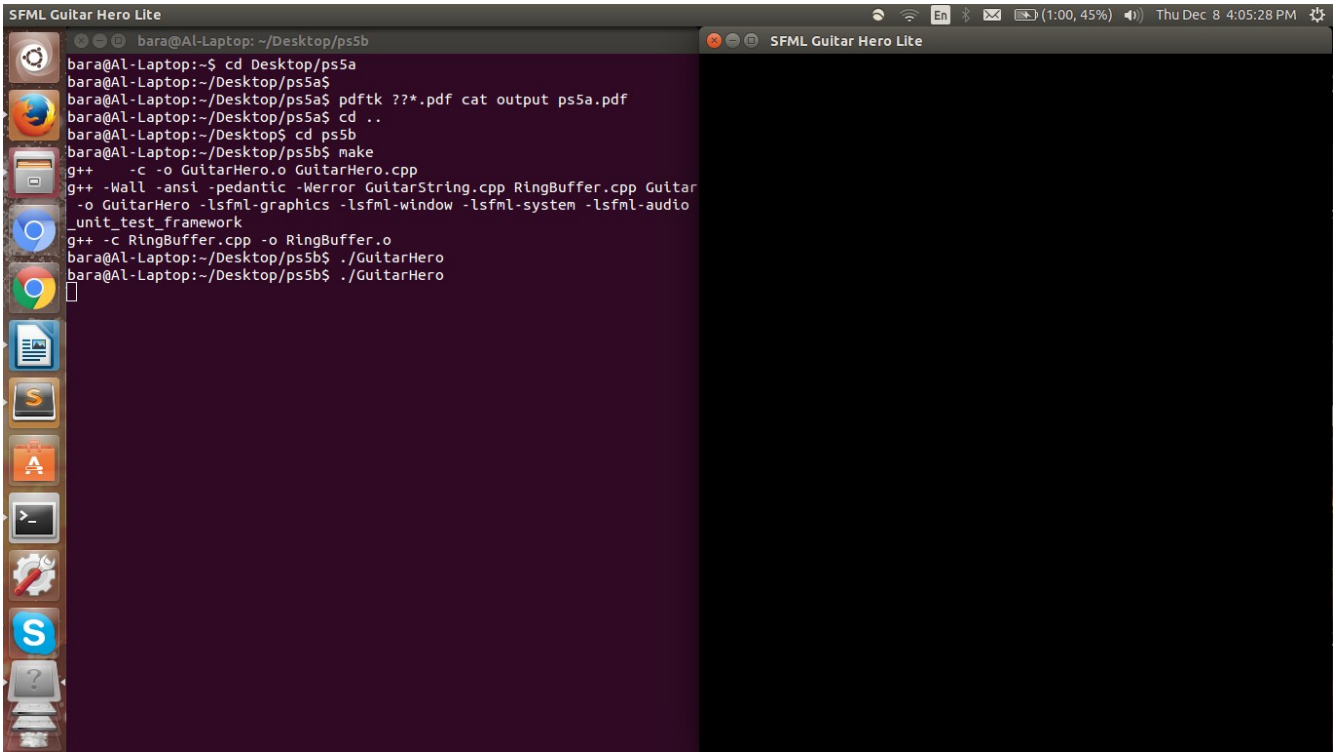
PS5b:

The main purpose of this assignment is to actually make the simulation of the guitar string. This part we are plucking of the string and inserting white noise into every element. We needed to be able to use keys to play sound from each key. To accomplish this task, we needed to first create the guitar string of the given frequency using the sampling rate of 44,100Hz. Then another constructor that takes in a vector of type int_16 (double). Which will initialize the contents of the buffer to the values. The create a function that replaces the items in the buffer with random values between -32768 to 32767. Then a function to delete the sample at the front of the ring buffer and add to the end of the ring buffer the average of the first two samples, multiplied by the energy decay factor.

In this program, I created a pointer to a ring buffer to have sounds plucked in the buffer. I created a member variable to save the sampling rate divided by the given frequency. The first constructor, I enqueued 0 in each element of the amount the user wants for the frequency. In the other constructor, it would be called vectors of type sf::Int16. It would allocated a Ring Buffer of the size given. In the pluck function, I deleted the current Ring Buffer, and relocated a new Ring Buffer. Then I made a loop to insert white noise to each element. In the tic function, I created a 2 variables of type double that save the first value, and return the front of the array. I then added both and divided by 2 to multiply it by the decay. Then finally enqueueing the result value into the ring buffer. Then in the main, I had each key becomes registered and given a sound. To do this, I used the SFML library to use the keyboard function.

The Guitar String assignment helped me understand how to create sound and input them into any key. In the future, maybe creating my own program that can simulate different types of instruments. More understanding of the use of inheritance and the use of calling a function while allocating memory.



(Compiled with makefile)
(Extra credit for layout was not created)

```
    1: cc = g++
    2:
    3: all : GuitarString RingBuffer GuitarHero
    4:
    5: GuitarString : GuitarString.o RingBuffer.o GuitarHero.o
    6:         $(cc) -Wall -ansi -pedantic -Werror GuitarString.cpp RingBuffer.cpp
GuitarHero.cpp -o GuitarHero -lsfml-graphics -lsfml-window -lsfml-system -lsfml-aud
io -l boost_unit_test_framework
    7:
    8: GuitarString.o : GuitarString.hpp RingBuffer.hpp
    9:         $(cc) -c GuitarString.cpp -o GuitarString.o
   10:
   11: RingBuffer : RingBuffer.hpp
   12:         $(cc) -c RingBuffer.cpp -o RingBuffer.o
   13:
   14: GuitarHero : GuitarHero.cpp RingBuffer.hpp GuitarString.hpp
   15:         $(cc) -c GuitarHero.cpp -o GuitarHero.o
   16:
   17:
   18: clean :
   19:         rm  *.o GuitarHero debug
   20:
   21: debug :
   22:         g++ -g -Wall -ansi -pedantic -Werror GuitarString.cpp RingBuffer.cpp
 GStest.cpp -o debug -l boost_unit_test_framework
   23:
```

```
 1: /*Copyright [2016] <Albara Mehene> */
 2: /*#define BOOST_TEST_DYN_LINK
 3: #define BOOST_TEST_MODULE Main
 4: */
 5:
 6: #define BOOST_TEST_DYN_LINK
 7: #define BOOST_TEST_MODULE Main
 8: #include <boost/test/unit_test.hpp>
 9:
10: #include <stdint.h>
11: #include <iostream>
12: #include <string>
13: #include <exception>
14: #include <stdexcept>
15:
16: #include "RingBuffer.hpp"
17:
18: BOOST_AUTO_TEST_CASE(RBcontructor) {
19:     // normal constructor
20:     BOOST_REQUIRE_NO_THROW(RingBuffer(100));
21:
22:     // this should fail
23:     BOOST_REQUIRE_THROW(RingBuffer(0), std::exception);
24:     BOOST_REQUIRE_THROW(RingBuffer(0), std::invalid_argument);
25: }
26:
27: BOOST_AUTO_TEST_CASE(RBenque_dequeue) {
28:     RingBuffer rb(100);
29:
30:     rb.enqueue(2);
31:     rb.enqueue(1);
32:     rb.enqueue(0);
33:
34:     BOOST_REQUIRE(rb.dequeue() == 2);
35:     BOOST_REQUIRE(rb.dequeue() == 1);
36:     BOOST_REQUIRE(rb.dequeue() == 0);
37:
38:     BOOST_REQUIRE_THROW(rb.dequeue(), std::runtime_error);
39: }
40:
41: BOOST_AUTO_TEST_CASE(test1) {
42:     RingBuffer rb(3);
43:     rb.enqueue(2);
44:     rb.enqueue(3);
45:     rb.enqueue(4);
46:     BOOST_REQUIRE(rb.isFull());
47:     BOOST_REQUIRE_THROW(rb.enqueue(5), std::runtime_error)
48: }
```

```
 1: /*Copyright [2016] <Albara Mehene> */
 2: #ifndef RINGBUFFER_HPP
 3: #define RINGBUFFER_HPP
 4:
 5: #include <boost/test/unit_test.hpp>
 6:
 7: #include <SFML/Graphics.hpp>
 8: #include <SFML/System.hpp>
 9: #include <SFML/Audio.hpp>
10: #include <SFML/Window.hpp>
11: #include <stdint.h>
12: #include <iostream>
13: #include <string>
14: #include <exception>
15: #include <stdexcept>
16: #include <vector>
17:
18: class RingBuffer{
19:  public:
20:     explicit RingBuffer(int capacity);
21:     int size();
22:     bool isEmpty();
23:     bool isFull();
24:     void enqueue(int16_t x);
25:     int16_t dequeue();
26:     int16_t peek();
27:     void print_out();
28:     ~RingBuffer();
29:  private:
30:     int16_t *first;
31:     int16_t *last;
32:     int16_t *array;
33:     int count;
34:     int cap;
35: };
36:
37: #endif
```

```
 1: /*Copyright [2016] <Albara Mehene> */
 2:
 3: #include "RingBuffer.hpp"
 4:
 5: // creates a empty ringbuffer with a max capacity
 6: RingBuffer::RingBuffer(int capacity) {
 7: if(capacity < 1) {
 8:     throw std::invalid_argument("capacity must be greater than zero");
 9: }
10:     cap = capacity;
11:     count = 0;
12:     array = new int16_t[capacity];
13:     first = array;
14:     last = array;
15: }
16:
17: // returns the number of items currently in the buffer
18: int RingBuffer::size() {
19:     return count;
20: }
21:
22: // checks to see if the buffer is empty
23: bool RingBuffer::isEmpty() {
24:     if (count == 0) {
25:         return 1;
26:     } else {
27:         return 0;
28:     }
29: }
30:
31: // checks to see if the buffer is full
32: bool RingBuffer::isFull() {
33:     if (count == cap) {
34:        return 1;
35:     } else {
36:        return 0;
37:     }
38: }
39:
40: // add item x to the end of buffer
41: void RingBuffer::enqueue(int16_t x) {
42: if(isFull() == 1) {
43:     throw std::runtime_error("can't enqueue to a full ring");
44: }
45:
46:     if (last == (array+(cap-1))){
47:          count++;
48:          (*last) = x;
49:          last = array;
50:     }else{
51:         count++;
52:         (*last) = x;
53:         last = (last + 1);
54:     }
55: }
56:
57: // deletes and returns the item from the front of the buffer
58: int16_t RingBuffer::dequeue() {
59:     if (isEmpty() == 1) {
60:        throw std::runtime_error("can't dequeue to a empty ring");
61:     }
```

```
62:
63:     int16_t store;
64:
65:     if(first == (array+(cap-1))){
66:       count--;
67:       store = (*first);
68:       first = array;
69:       return store;
70:     }else{
71:         count--;
72:         store = (*first);
73:         first = (first+1);
74:         return store;
75:     }
76: }
77:
78: // returns item from the front without deleting it
79: int16_t RingBuffer::peek() {
80:     int16_t temp = (*first);
81:     return temp;
82: }
83:
84: RingBuffer::~RingBuffer() {
85:     delete[] array;
86: }
87: void RingBuffer::print_out(){
88:
89:     for(int i = 0; i < cap;i++){
90:         std::cout << "Array:   " << array[i] << std::endl;
91:     }
92: }
```

```
    1: /*Copyright Albara Mehene*/
    2: #include <SFML/Graphics.hpp>
    3: #include <SFML/System.hpp>
    4: #include <SFML/Audio.hpp>
    5: #include <SFML/Window.hpp>
    6:
    7: #include <math.h>
    8: #include <limits.h>
    9:
   10: #include <iostream>
   11: #include <string>
   12: #include <exception>
   13: #include <stdexcept>
   14: #include <vector>
   15:
   16: #include "RingBuffer.hpp"
   17: #include "GuitarString.hpp"
   18:
   19: #define SAMPLES_PER_SEC 44100.0
   20: #define SAMPLE 37
   21:
   22: std::vector<sf::Int16> makeSamplesFromString(GuitarString &gs)  {
   23:   std::vector<sf::Int16> samples;
   24:   int duration = 8;
   25:   gs.pluck();
   26:   int i;
   27:   for (i= 0; i < SAMPLES_PER_SEC * duration; i++) {
   28:     gs.tic();
   29:     samples.push_back(gs.sample());
   30:   }
   31:
   32:   return samples;
   33: }
   34:
   35:
   36:
   37: int main() {
   38:   sf::RenderWindow window(sf::VideoMode(300, 200), "SFML Guitar Hero Lite");
   39:   sf::Event event;
   40:
   41:   std::vector < std::vector<sf::Int16> > sample(SAMPLE);
   42:   std::vector <sf::Sound> sound(SAMPLE);
   43:   std::vector <sf::SoundBuffer> buffer(SAMPLE);
   44:   std::string keyboard = ("1234567890qwertyuiopasdfghjklzxcvbnm,");
   45:
   46:   // inserts all sounds in the buffer
   47:   for (int i = 0; i < SAMPLE; i++) {
   48:     GuitarString GStemp(440.0 * pow(2, (i - 24)/12.0));
   49:     sample[i] = makeSamplesFromString(GStemp);
   50:     if (!(buffer[i].loadFromSamples(&(sample[i][0]), sample[i].size(), 2 , 4
4100.0))) {
   51:      throw std::runtime_error(" sf::SoundBuffer: failed to load from sample.
 ");
   52:     }
   53:     sound[i].setBuffer(buffer[i]);
   54:   }
   55:
   56:   while (window.isOpen()) {
   57:     while (window.pollEvent(event)) {
   58:       switch (event.type) {
   59:         case sf::Event::Closed:
```

```
60:           window.close();
61:           break;
62:         default:
63:         if (sf::Event::KeyPressed && event.key.code != -1) {
64:           int Key = keyboard.find(event.key.code);
65:           sound[Key].play();
66:         }
67:           break;
68:       }
69:       window.clear();
70:       window.display();
71:     }
72:   }
73:   return 0;
74: }
```

```
 1:
 2: #define BOOST_TEST_DYN_LINK
 3: #define BOOST_TEST_MODULE Main
 4: #include <boost/test/unit_test.hpp>
 5:
 6: #include <vector>
 7: #include <exception>
 8: #include <stdexcept>
 9:
10:
11: #include "GuitarString.hpp"
12:
13: BOOST_AUTO_TEST_CASE(GS) {
14:   std::vector <sf::Int16> v;
15:
16:   v.push_back(0);
17:   v.push_back(2000);
18:   v.push_back(4000);
19:   v.push_back(-10000);
20:
21:   BOOST_REQUIRE_NO_THROW(GuitarString gs = GuitarString(v));
22:
23:   GuitarString gs = GuitarString(v);
24:
25:   // GS is 0 2000 4000 -10000
26:   BOOST_REQUIRE(gs.sample() == 0);
27:
28:   gs.tic();
29:   // it's now 2000 4000 -10000 996
30:   BOOST_REQUIRE(gs.sample() == 2000);
31:
32:   gs.tic();
33:   // it's now 4000 -10000 996 2988
34:   BOOST_REQUIRE(gs.sample() == 4000);
35:
36:   gs.tic();
37:   // it's now -10000 996 2988 -2988
38:   BOOST_REQUIRE(gs.sample() == -10000);
39:
40:   gs.tic();
41:   // it's now 996 2988 -2988 -4483
42:   BOOST_REQUIRE(gs.sample() == 996);
43:
44:   gs.tic();
45:   // it's now 2988 -2988 -4483 1984
46:   BOOST_REQUIRE(gs.sample() == 2988);
47:
48:   gs.tic();
49:   // it's now -2988 -4483 1984 0
50:   BOOST_REQUIRE(gs.sample() == -2988);
51:
52:   // a few more times
53:   gs.tic();
54:   BOOST_REQUIRE(gs.sample() == -4483);
55:   gs.tic();
56:   BOOST_REQUIRE(gs.sample() == 1984);
57:   gs.tic();
58:   BOOST_REQUIRE(gs.sample() == 0);
59: }
```

```
 1: /*Copyright [2016] <Albara Mehene> */
 2: #ifndef GUITARSTRING_H
 3: #define GUITARSTRING_H
 4:
 5: #include <iostream>
 6: #include <string>
 7: #include <exception>
 8: #include <stdexcept>
 9: #include <vector>
10: #include <cmath>
11:
12: #include "RingBuffer.hpp"
13:
14: const double DECAY = 0.996;
15: const double SAMPLING_RATE = 44100;
16:
17: class GuitarString{
18:  public:
19:     explicit GuitarString(double frequency);
20:     explicit GuitarString(std::vector <sf::Int16> init);
21:     GuitarString();
22:     void pluck();
23:     void tic();
24:     sf::Int16 sample();
25:     int time();
26:     ~GuitarString();
27:  private:
28:     RingBuffer *_rb;
29:     int _ticNum;
30:     int G_cap;
31: };
32:
33: #endif
```

```
    1: #include "GuitarString.hpp"
    2:
    3: /*create a guitar string of the given frequency using a
    4: sampling rate of 44,100.*/
    5: GuitarString::GuitarString(double frequency){
    6:         G_cap = (ceil(SAMPLING_RATE/frequency));
    7:
    8:         this->_rb = new RingBuffer(G_cap);
    9:
   10:         //fill with 0s to represent the guitar string at rest
   11:         for(int i = 0; i < G_cap; i++){
   12:                 this->_rb->enqueue(0);
   13:         }
   14: }
   15:
   16: /*create a guitar string with size and initial values are given
   17: by the vector*/
   18: GuitarString::GuitarString(std::vector <sf::Int16> init){
   19:
   20:         this->_rb = new RingBuffer(init.size());
   21:
   22:         for (std::vector<sf::Int16>::iterator i = init.begin(); i != init.en
d(); ++i){
   23:       _rb->enqueue(*i);
   24:
   25:         }
   26: }
   27:
   28: GuitarString::GuitarString(){
   29:
   30: }
   31:
   32: /*Pluck the guitar string by replacing the buffer with random
   33: values, representing white noise*/
   34: void GuitarString::pluck(){
   35:                 //empty the buffer
   36:                 delete _rb;
   37:                 _rb = new RingBuffer(G_cap);
   38:
   39:                 //replace with random noise (white noise)
   40:                 for(int i = 0; i < G_cap; ++i){
   41:
   42:                         _rb->enqueue((int16_t)(rand() & 0xffff));
   43:                 }
   44:
   45: }
   46:
   47: /*advance the simulation one time step */
   48: void GuitarString::tic(){
   49:         _ticNum++;
   50:
   51:         double first = _rb->dequeue();
   52:         double front = sample();
   53:
   54:         //std::cout << "Decay: " << DECAY << std::endl;
   55:
   56:         double value = DECAY * ((first + front)/2);
   57:
   58:         //std::cout << "value:   " << value << std::endl;c;
   59:         _rb->enqueue(value);
   60: }
```

```
61: /*return the current sample*/
62: sf::Int16 GuitarString::sample(){
63:         //May be returning this incorrectly
64:         return _rb->peek();
65: }
66:
67: /*return number of times tic was called so far*/
68: int GuitarString::time(){
69:
70:         return _ticNum;
71: }
72:
73: GuitarString::~GuitarString(){
74:         //need to use delete function
75:         delete _rb;
76:
77: }
```