Thomas Freeman

Computing IV: Project Portfolio

Fall Semester 2021

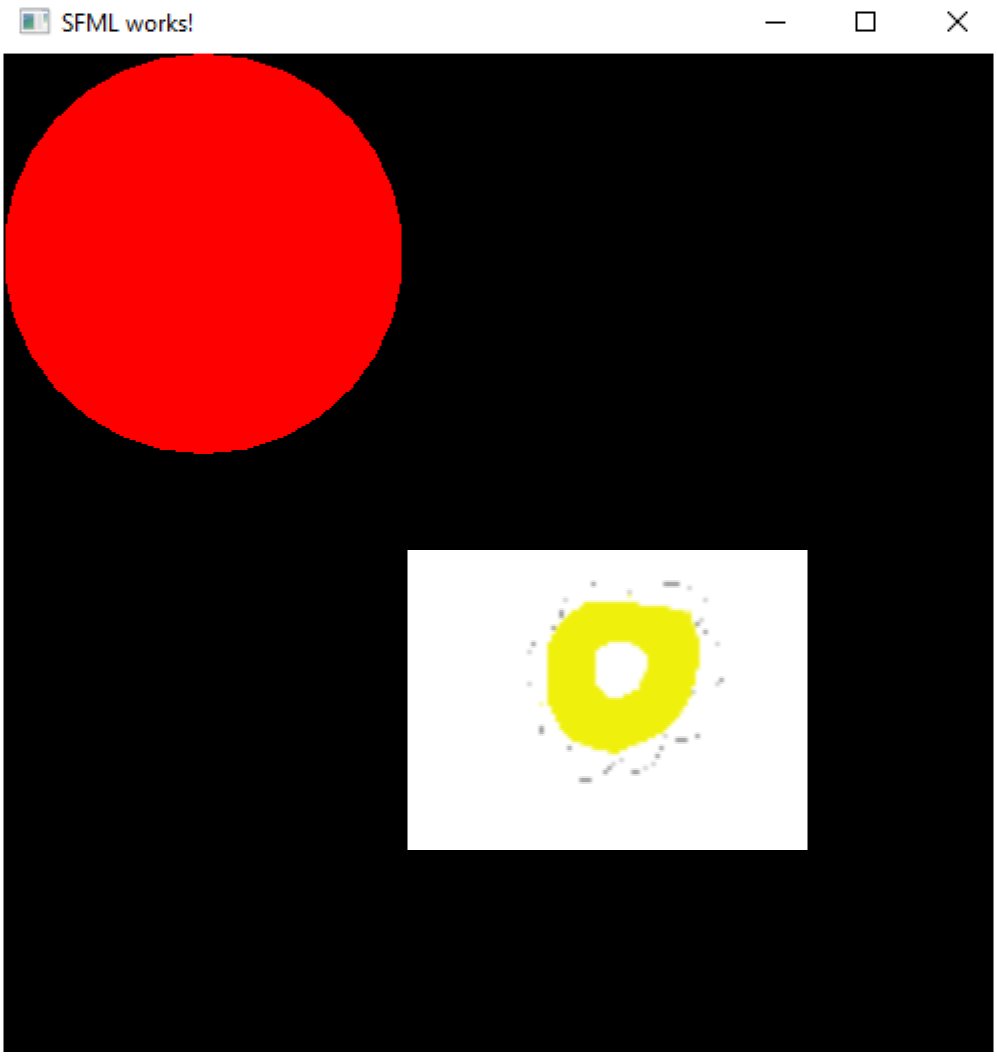
Contents:

PS0 – Hello World with SFML

The goal of our first assignment was to get us used to using the SFML libraries with our c++ code, as well as giving me some insight on how we would write our expressions and variables for future assignments. We are given most of the code for this assignment already, as it’s part of the setup for the SFML libraries. All we needed to do was simply modify the given code so that we could create our own sprite (I.e any .PNG file) and have it move around the screen. For extra credit we could implement an additional feature, What I ended up doing for an extra feature is binding the key binds for the sprite's movement with a specific color in the SFML libraries Up is associated with Blue, the down arrow is associated with red, etc.

The following code is associated with PS0

```
//Thomas Freeman
// implementation of main.cpp
// 9/13/21


#include <SFML/Graphics.hpp>
#include <iostream>

int main() {

    // Create the window
    sf::RenderWindow window(sf::VideoMode(500, 500), "SFML works!");

    // Load a sprite
    sf::Texture texture;
    if (!texture.loadFromFile("sprite.png"))
        return EXIT_FAILURE;
    sf::Sprite sprite(texture);

    // Create a circle
    sf::CircleShape shape(100.f);
    shape.setFillColor(sf::Color::Blue);



    while (window.isOpen()) {
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed)
                window.close();


        }

        window.clear();




        float offsetX = 0;
        float offsetY = 0;

        // Get current position of the sprite
        sf::Vector2f pos = sprite.getPosition();

        // Moves the sprite around
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left) && pos.x != 0)
        offsetX = -1;
    else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right) && pos.x != 400 - 198)
            offsetX = 1;
    else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up) && pos.y != 0)
            offsetY = -1;
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down) && pos.y != 400 - 152)
            offsetY = 1;
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
            shape.setFillColor(sf::Color::Magenta);
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
            shape.setFillColor(sf::Color::Yellow);
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up))
            shape.setFillColor(sf::Color::Blue);
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down))
            shape.setFillColor(sf::Color::Red);
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::R)) {
            sprite.setPosition(0, 0);
            pos.x = pos.y = 0;
        }
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape))
            window.close();

        // Assign it to a new position
        sprite.setPosition(pos.x + offsetX, pos.y + offsetY);

        // Draw the images
        window.draw(shape);
        window.draw(sprite);

        window.display();
    }

    return 0;
}
```



PS1 Linear Feedback Shift Register and Image Encoding

This was our first two-part assignment so I will explain my solution for the first part. To begin PS1, we needed to create our Linear Feedback shift register, operating with 16-bit Fibonacci as our given linear function. There are two methods that are implemented in this first part. Firstly, is the step function, which simulates one step of the LFSR given a seed to use. Generate is the other, simulating the step function k times such that necessary arithmetic of multiple steps is shown at the end. The output of this program is a dynamic array of integers (1’s and 0’s) that show the steps that came out of the generate function. This assignment was also our introduction to using the boost libraries in order to test certain outputs in our code. I tested for if a given string was 16 bits in length and if the input was properly done (I.e without spaces, unnecessary characters, etc). If either of these tests fail, errors are thrown.

The following is code for ps1a

```
CC = g++
CFLAGS = -std=c++11 -c -g -Og -Wall -Werror -pedantic
LFLAGS = -lboost_unit_test_framework
OBJS = test.o FibLFSR.o
EXE = ps1a
all : $(EXE)
ps1a : $(OBJS)
	$(CC) $(OBJS) -o $(EXE) $(LFLAGS)
test.o : test.cpp
	$(CC) $(CFLAGS) test.cpp
FibLFSR.o : FibLFSR.cpp FibLFSR.h
	$(CC) $(CFLAGS) FibLFSR.cpp
clean :
	\rm *.o $(EXE)
```

```
// Thomas Freeman
// test.cpp for PS1a
// Provided test file that I modified with my own Boost tests
#include <iostream>
#include <string>
#include "FibLFSR.h"
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Main
#include <boost/test/unit_test.hpp>
BOOST_AUTO_TEST_CASE(sixteenBitsThreeTaps) {
   FibLFSR l("1011011000110110");
   BOOST_REQUIRE(l.step() == 0);
   BOOST_REQUIRE(l.step() == 0);
   BOOST_REQUIRE(l.step() == 0);
   BOOST_REQUIRE(l.step() == 1);
   BOOST_REQUIRE(l.step() == 1);
   BOOST_REQUIRE(l.step() == 0);
   BOOST_REQUIRE(l.step() == 0);
   BOOST_REQUIRE(l.step() == 1);
   FibLFSR l2("1011011000110110");
   BOOST_REQUIRE(l2.generate(9) == 51);
}
// First test case shows simple input and output using the
// test value from the initial test case.
BOOST_AUTO_TEST_CASE(my_test1) {

   std::cout << "First Test" << std::endl;
   FibLFSR lfsr("1011011000110110");
   std::cout << " Original seed: " << lfsr << std::endl;

   int result = lfsr.generate(5);
   BOOST_REQUIRE(result == 3);
```

```

  std::cout << "After generate(5): " << lfsr << " " << result <<
std::endl;
  std::cout << std::endl;
}

  // This test is meant to show when an input seed
  // is too long or too short.
  BOOST_AUTO_TEST_CASE(my_test2) {

      std::cout << "\n Test Case #2" << std::endl;
      std::string testSeed1 = "111000110";
      std::string testSeed2 = "0000000000000000001";

      std::cout << "Seed is less than 16 bits : "
          << std::endl;
      BOOST_REQUIRE_THROW(FibLFSR("0101100101"),
std::invalid_argument);

      std::cout << "Seed is more than 16 bits :  "
          "0111011000110110101100" << std::endl;
      BOOST_REQUIRE_THROW(FibLFSR("0111011000110110101100"),
          std::invalid_argument);
  }
```

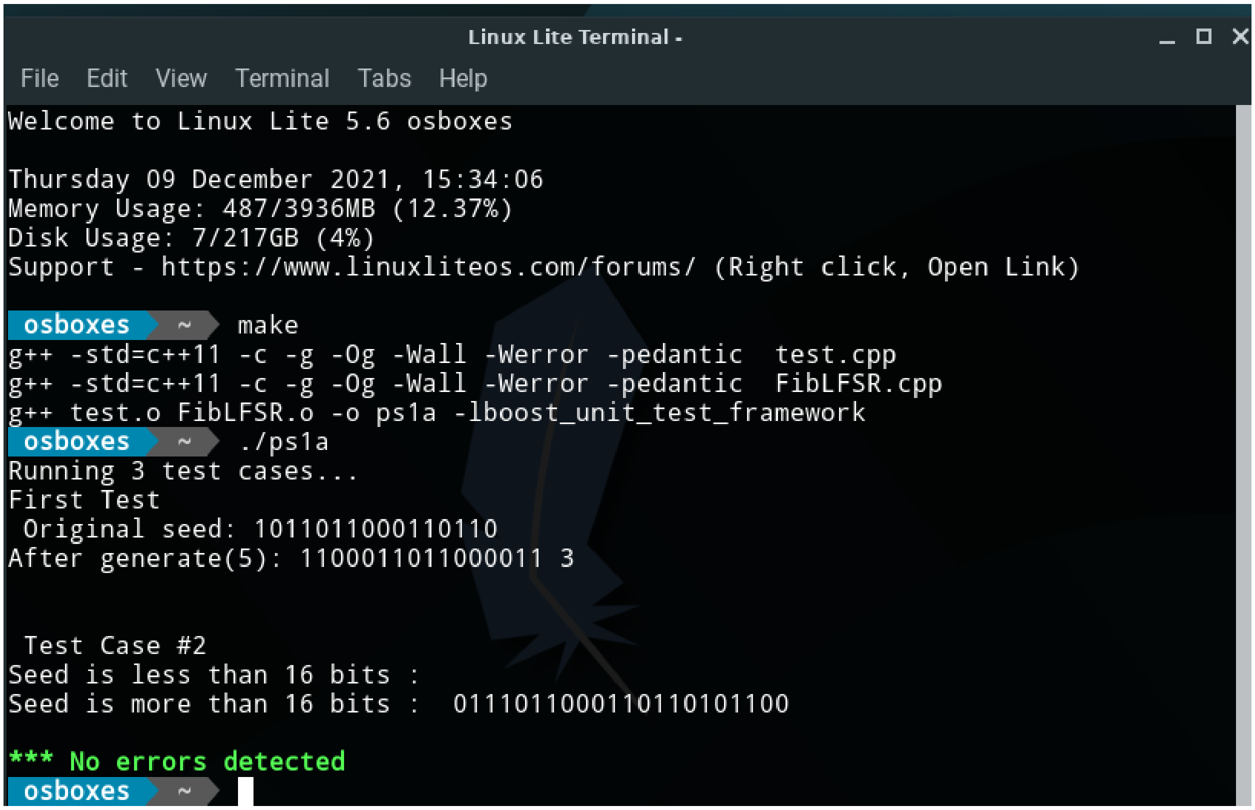
```
// Thomas Freeman
// implementation of FibLFSR.h
// Simple header file to prevent building
// this register from becoming a nightmare.



#ifndef FIBLFSR_H
#define FIBLFSR_H
#include <iostream>
#include <exception>

class FibLFSR {
public:
  FibLFSR(std::string seed);
  //Constructor creates LFSR with a given seed and set of tap bits

  explicit FibLFSR(FibLFSR& copyLFSR);
  explicit FibLFSR(FibLFSR&& moveLFSR) noexcept;
  ~FibLSFR();
  FibLFSR& operator=(const FibLFSR& rvalue);
  FibLFSR& operator=(FibLFSR&& rvalue) noexcept;

  int step();            // Shows one step in the algorithm and
returns a bit
```

```
   int generate(int k);   // Shows a given number of steps (k) and
returns a value.

   // Displays the register
   friend std::ostream& operator<<(std::ostream& out, const FibLFSR&
flfsr);

private:
   int size;               // Register size
   int* reg;               // Defines register as an array of integers
};
#endif
```

```
// Thomas Freeman
// implementation of FibLFSR.cpp
// This code is what builds and allows the
// sixteen bit linear feedback register to perform operations
// along with allowing tests to be used via the boost library's.
// I did this by using an array to store the string of bits and
// wrote out the step and generate functions.


#include "FibLFSR.h"
#define ASCII_OFFSET 48

// Our FibLFSR constructor, uses a 16 character
// string as input
FibLFSR::FibLFSR(std::string seed) {

   size = seed.length();
   if (size == 16) {
   reg = new int[size];

         int count = 0;
         for (int i = size - 1; i >= 0; i--) {
         reg[i] = (int)seed[count] - ASCII_OFFSET;
         count++;
         }
   }

   else {

         reg = nullptr;
         size = 0;
         //No try-catch block for BOOST tests
         throw std::invalid_argument(
               "Seeds are sixteen bits only!");
   }
}

FibLFSR::FibLFSR(FibLFSR& copyLFSR) {

   size = copyLFSR.size;
   if (size > 0) {
   reg = new int[size];
         for (int i = 0; i < size; i++) {
         reg[i] = copyLFSR.reg[i];
         }
   }
   else reg = nullptr;
}

FibLFSR::FibLFSR(FibLFSR&& moveLFSR) noexcept {

   size = moveLFSR.size;
   reg = moveLFSR.reg;
   moveLFSR.reg = nullptr;
   moveLFSR.size = 0;
}

FibLFSR::~FibLFSR() {

   if (reg != nullptr) delete[] reg;
   reg = nullptr;
   size = 0;
}


// The Assignment operators for
// Assigning new values.
FibLFSR& FibLFSR::operator=(const FibLFSR& rvalue) {

   if (this == &rvalue) return *this;
   if (reg != nullptr) delete[] reg;

   size = rvalue.size;
   if (size > 0) {
   reg = new int[size];
   for (int i = 0; i < size; i++) {
         reg[i] = rvalue.reg[i];
         }
   }
   return *this;
}

FibLFSR& FibLFSR::operator=(FibLFSR&& rvalue) noexcept {

   if (this == &rvalue) return *this;
   if (reg != nullptr) delete[] reg;

   size = rvalue.size;
   reg = rvalue.reg;
   rvalue.reg = nullptr;
   rvalue.size = 0;

```

```
  return *this;
}


// Step allows us to show one step of the register, specifically the
// exclusive or operation on the leftmost bit. Returning the rightmost
bit.
int FibLFSR::step() {

  int rightmostBit = reg[15] ^ reg[13];
  rightmostBit ^= reg[12];
  rightmostBit ^= reg[10];

  // shifts each bit left except for the rightmost bit
  for (int i = size - 1; i >= 1; i--) {
        reg[i] = reg[i - 1];
  }
  reg[0] = rightmostBit;
  // As a result, we create a new rightmost bit

  return rightmostBit;
}


// Generate takes an int K and generates k number of steps for the
LFSR to produce int k
int FibLFSR::generate(int k) {

  int result = 0;
  for (int i = k; i > 0; i--) {
        result *= 2;
        result += step();
  }
  return result;
}

// Operator displays bits from highest to lowest index by overloading
the stream insertion operator
std::ostream& operator<<(std::ostream& out, const FibLFSR& flfsr) {

  for (int i = flfsr.size - 1; i >= 0; i--) {
        out << flfsr.reg[i];
  }
  return out;
}
```

All tests are ran with the same seed



```
Welcome to Linux Lite 5.6 osboxes

Thursday 09 December 2021, 15:34:06
Memory Usage: 487/3936MB (12.37%)
Disk Usage: 7/217GB (4%)
Support - https://www.linuxliteos.com/forums/ (Right click, Open Link)

osboxes ~  make
g++ -std=c++11 -c -g -Og -Wall -Werror -pedantic  test.cpp
g++ -std=c++11 -c -g -Og -Wall -Werror -pedantic  FibLFSR.cpp
g++ test.o FibLFSR.o -o ps1a -lboost_unit_test_framework
osboxes ~  ./ps1a
Running 3 test cases...
First Test
 Original seed: 1011011000110110
After generate(5): 1100011011000011 3


 Test Case #2
Seed is less than 16 bits :
Seed is more than 16 bits :  01110110001101101011100

*** No errors detected
osboxes ~
```

As for the second part of PS1, we needed to encode any image we wanted by implementing a new class to complement our LFSR. This new class contains a method called transform that takes a 16-bit seed and a given image and encodes said image via the step and generate functions. In order to get our outputs, we'd overload the stream operator to allow input and output to be done with our make files, becoming vital for future assignments. Eventually outputting two windows with the image before and after the encryption is done. This process can also be done in reverse taking an encrypted image and decrypting it to become normal again as well.

Here's the following code for ps1b

```
CC = g++
CFLAGS = -std=c++11 -c -g -Og
LIBS = -lsfml-graphics -lsfml-window -lsfml-system
OBJS = PhotoMagic.o FibLFSR.o
EXE = PhotoMagic
all : $(EXE)
PhotoMagic : $(OBJS)
    $(CC) $(OBJS) -o $(EXE) $(LIBS)
FibLFSR.o : FibLFSR.cpp FibLFSR.h
    $(CC) $(CFLAGS) FibLFSR.cpp
PhotoMagic.o : PhotoMagic.cpp
```

```
	$(CC) $(CFLAGS) PhotoMagic.cpp
clean :
	\rm *.o $(EXE)
```

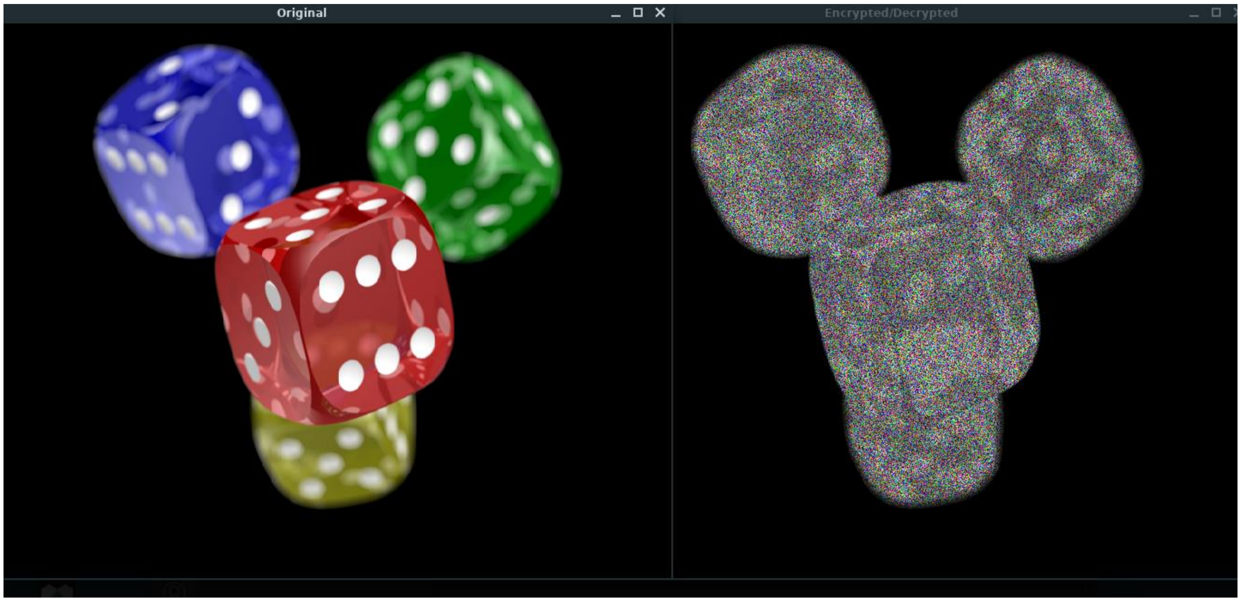
```
//Thomas Freeman
//implementation of PhotoMagic.cpp
//Main method linked to FibLFSR that
//transforms the image using two file
//and a 16 bit input via the command line.
#include "FibLFSR.h"
#include <SFML/System.hpp>
#include <SFML/Window.hpp>
#include <SFML/Graphics.hpp>
//The register is responsible for transforming the image
void transform(sf::Image& img, FibLFSR* flfsr);
int main(int argc, char* argv[]) {
   if (argc != 4) {
       std::cerr << "Usage: " << argv[0] <<
       " <input file> <output file> <16 bit binary seed>" << std::endl;
       return -1;
   }

   FibLFSR flfsr;
   try {
       flfsr = FibLFSR(argv[3]);
   }
   catch (std::invalid_argument invalid ) {
       std::cout << invalid.what() << std::endl;
       return -1;
   }
   //Shows our two windows
   sf::Image image;
   if (!image.loadFromFile(argv[1])) { return -1; }
   sf::Vector2u size = image.getSize();
   sf::RenderWindow window1(sf::VideoMode(size.x, size.y), "Original");
   sf::RenderWindow window2(sf::VideoMode(size.x, size.y), "Encrypted/"
   "Decrypted");
   //displays the images
   sf::Texture texture1;
   texture1.loadFromImage(image);
   sf::Sprite sprite1;
   sprite1.setTexture(texture1);
   transform(image, &flfsr);     //calls transform for the image file
   sf::Texture texture2;
   texture2.loadFromImage(image);
   sf::Sprite sprite2;
   sprite2.setTexture(texture2);
   // Allows both windows to be open at the same time
   while (window1.isOpen() && window2.isOpen()) {
       sf::Event event;
       while (window1.pollEvent(event)) {
            if (event.type == sf::Event::Closed) { window1.close(); }
        }
        while (window2.pollEvent(event)) {
            if (event.type == sf::Event::Closed) { window2.close(); }
        }
        window1.clear();
        window1.draw(sprite1);
        window1.display();
        window2.clear();
        window2.draw(sprite2);
        window2.display();
    }
    if (!image.saveToFile(argv[2])) { return -1; }
    return 0;
 }
 //Implementation of the transform method to alter the pixels
void transform(sf::Image& img, FibLFSR* flfsr) {
    sf::Vector2u size = img.getSize();
    sf::Color p;        //p is for pixel
    for (int x = 0; x < (int)size.x; x++) {
        for (int y = 0; y < (int)size.y; y++) {
            p = img.getPixel(x, y);
            p.r ^= flfsr->generate(8);
            p.g ^= flfsr->generate(8);
            p.b ^= flfsr->generate(8);
            img.setPixel(x, y, p);
        }
    }
 }
```

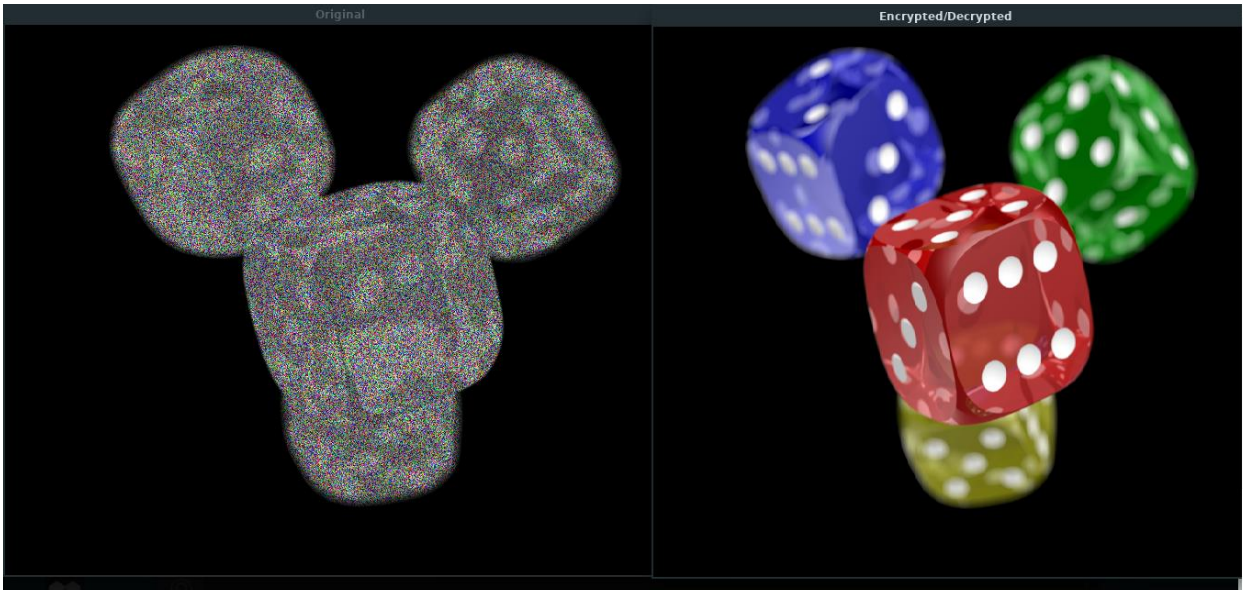
```
//Thomas Freeman
//Implementation of FibLFSR.cpp header file
//Sets up and displays all methods like ps1a
#include <iostream>
#include <exception>
#include <math.h>
#ifndef FIBLFSR_H
#define FIBLFSR_H
#define ASCII_OFFSET 48
class FibLFSR {
public:
   FibLFSR();
   FibLFSR(std::string seed);  // Constructor for the register
                               // with the seed and tap
   explicit FibLFSR(const FibLFSR& copyFibLFSR);
   explicit FibLFSR(FibLFSR&& moveFibLFSR) noexcept;
   ~FibLFSR();
   FibLFSR& operator=(const FibLFSR& rvalue);
   FibLFSR& operator=(FibLFSR&& rvalue) noexcept;
   int step();             // simulates a step of the register

   int generate(int k);    // simulates k steps of the register
   //Register is displayed
   friend std::ostream& operator<<(std::ostream& out, const FibLFSR&
   flfsr);
private:
   int size;               // register size
   int* reg;               // integers in an array for the register
};
#endif
```

```
//Thomas Freeman
//Implementation of FibLFSR.cpp
//Tweaked from the first part to allow for
//the intended tap bits to work with the register.
//this allows each shift to help in encrypting the
//given image.
#include "FibLFSR.h"
#define TAP_BIT1 13
#define TAP_BIT2 12
#define TAP_BIT3 10
//The Constructors
FibLFSR::FibLFSR() : size(0), reg(nullptr) {}
FibLFSR::FibLFSR(std::string seed) {
   size = seed.length();
   if (size == 16) {
   reg = new int[size];
   int bit = 0;
   int count = 0;
       for (int i = size - 1; i >= 0; i--) {

       bit = (int)seed[count] - ASCII_OFFSET;
           if (bit == 0 || bit == 1) {
           reg[i] = bit;
           count++;
           }
           else {
               reg = nullptr;
               size = 0;
            throw std::invalid_argument("Incorrect seed, seed must be"
               " in binary form");
       }
   }
   }
   //Throws exception for strings > 16 bits
   else {
       reg = nullptr;
       size = 0;
    throw std::invalid_argument("Incorrect seed size, seed must be 16"
        " bits long");
}
}
```

```
//Most of the code doesn't change in this method
FibLFSR::FibLFSR(const FibLFSR& copyLFSR) {
   size = copyLFSR.size;
   if (size > 0) {
   reg = new int[size];
   for (int i = 0; i < size; i++) {
       reg[i] = copyLFSR.reg[i];
   }
   }
   else reg = nullptr;
}
FibLFSR::FibLFSR(FibLFSR&& moveLFSR) noexcept {
   size = moveLFSR.size;
   reg = moveLFSR.reg;
   moveLFSR.reg = nullptr;
   moveLFSR.size = 0;
}
FibLFSR::~FibLFSR() {
   if (reg != nullptr) delete[] reg;
   reg = nullptr;
   size = 0;
}
//Assignment operators need to be overloaded like lastime
FibLFSR& FibLFSR::operator=(const FibLFSR& rvalue) {
   if (this == &rvalue) return *this;
   if (reg != nullptr) delete[] reg;
   size = rvalue.size;
   if (size > 0) {
   reg = new int[size];
   for (int i = 0; i < size; i++) {
   reg[i] = rvalue.reg[i];
   }
   }
   return *this;
}
FibLFSR& FibLFSR::operator=(FibLFSR&& rvalue) noexcept {
   if (this == &rvalue) return *this;
   if (reg != nullptr) delete[] reg;
   size = rvalue.size;
   reg = rvalue.reg;
   rvalue.reg = nullptr;
   rvalue.size = 0;
   return *this;
}
//Substitute the Tap bits into the step function
int FibLFSR::step() {
   int rightmostBit = reg[size - 1] ^ reg[TAP_BIT1];
   rightmostBit ^= reg[TAP_BIT2];
   rightmostBit ^= reg[TAP_BIT3];
   //Still shifts the bits from left to right
   for (int i = size - 1; i >= 1; i--) {
   reg[i] = reg[i - 1];
   }
   reg[0] = rightmostBit;  //assigns the rightmost bit
```

```
    return rightmostBit;
}
//generate method remains the same
int FibLFSR::generate(int k) {
    int result = 0;
    for (int i = k; i > 0; i--) {
    result *= 2;
    result += step();
    }
    return result;
}
// Stream insertion operator doesn't change again
std::ostream& operator<<(std::ostream& out, const FibLFSR& flfsr) {
    for (int i = flfsr.size - 1; i >= 0; i--) {
    out << flfsr.reg[i];
}
    return out;
}
```





PS2 – N-Body Simulation

This two-part assignment deals with simulating our own universe given some assets to use and text files to create simulation. For the first part, we just implemented a static universe into a single universe with our given assets. In order to output this, we overload the stream operator again to have the terminal determine input and output, since both classes are going to be drawable to the same window, we end up overriding the virtual draw function in our classes Celestial Body and Universe, allowing them to be drawn in the same window, otherwise they'd be in two different windows. I even did the extra credit for this assignment by implementing my own background (.jpg) called cosmos. How are the assets used? Well our input text files contain the number of objects in the universe on the first line and the radius of the universe on the second, with the rest of the file containing information on the planets, such as position based on x and y, and their mass. This positional data is stored as a vector in the universe class to ensure the planets are displayed in the right position. Velocity was included as well but this is only important for ps2b.

The following is the code for ps2a

```
CC = g++
CFLAGS = -std=c++11 -c -g
OBJS = main.o CelestialBody.o Universe.o
LIBS = -lsfml-graphics -lsfml-window -lsfml-system
EXE = NBody
all: $(EXE)
NBody: $(OBJS)
```

```
    $(CC) $(OBJS) -o $(EXE) $(LIBS)
%.o: %.cpp
    $(CC) $(CFLAGS) -o $@ $<
clean:
    \rm $(OBJS) $(EXE)
```

```
//Thomas Freeman
//Implementation of main.cpp
//main deals with the command line,
//providing our necessary command and the data
//from our input
#include "Universe.h"
int main(int argc, char* argv[]) {
   if (argc != 1) {
       std::cerr << "Incorrect usage. Usage: ./NBody < filename.txt" <<
std::endl;
       exit(-1);
   }
   Universe universe;
   try {
       universe = Universe("nbody/cosmos.jpg");
   }
   catch (std::invalid_argument err) {
       std::cout << err.what() << std::endl;
       exit(-1);
   }
   sf::Vector2u windowSize = universe.getImage().getSize();
   sf::RenderWindow window(sf::VideoMode(windowSize.x, windowSize.y),
"The Solar System!");
  //call universe for file inputs
   std::cin >> universe;
   //SFML window display
   while (window.isOpen()) {
       sf::Event event;
       while (window.pollEvent(event)) {
           if (event.type == sf::Event::Closed) { window.close(); }
       }
       window.clear();
       window.draw(universe);
       for (int i = 0; i < universe.getNumOfPlanets(); i++) {
           window.draw(*universe.planets[i]);
       }
       window.display();
   }
   return 0;
}
```

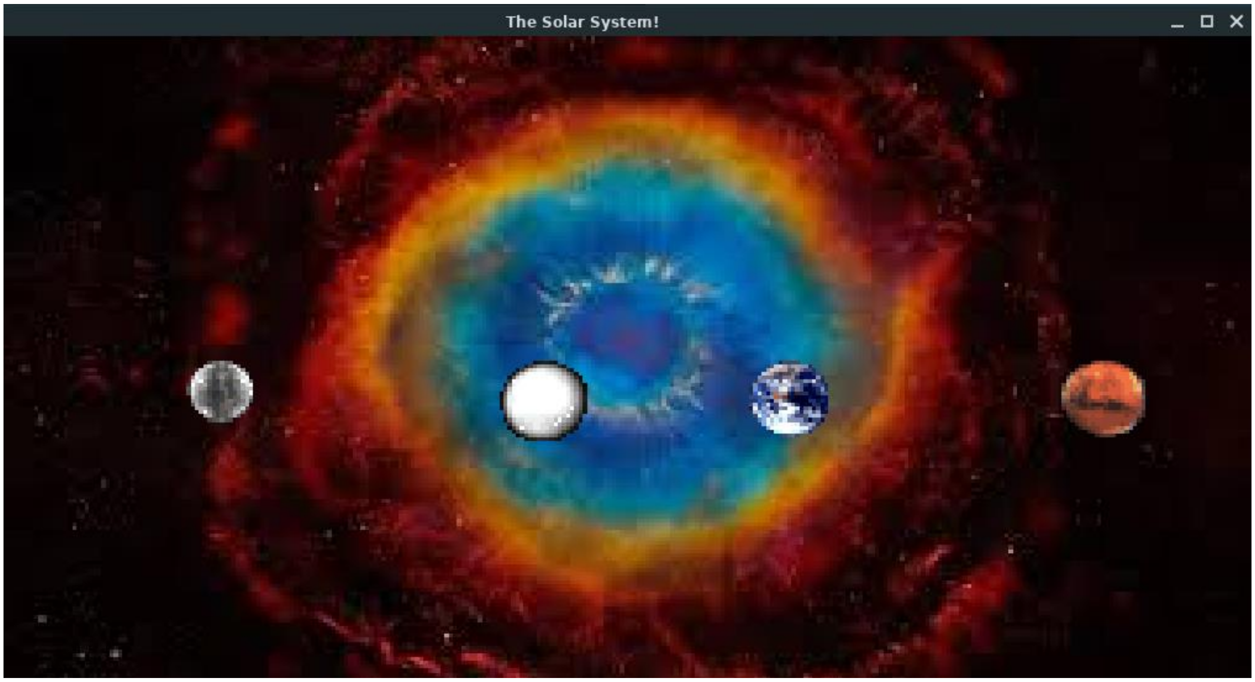
```
#ifndef CELESTIALBODY_H
#define CELESTIALBODY_H
#include <iostream>
#include <exception>
#include <SFML/Graphics.hpp>
class CelestialBody : public sf::Drawable {
public:
   // Necessary constructors
   CelestialBody();
   CelestialBody(float x_position, float y_position, float x_velocity,
   float y_velocity, float init_mass, std::string image_filename);
   //accessors and mutators
   float getXPos() const;
   float getYPos() const;
   float getXVel() const;
   float getYVel() const;
   float getMass() const;
   sf::Image getImage() const;
   void setXPos(float xPos);
   void setYPos(float yPos);
   void setXVel(float xVel);
   void setYVel(float yVel);
   void setMass(float mass);
   void setImage(std::string image_filename);
   // updates the position of a given planet
   void updatePosition();
private:
   //override virtual draw function
   void draw(sf::RenderTarget& target, sf::RenderStates states) const
override;
 float xPos, yPos;       //positions of x and y
   float xVel, yVel;       //Velocity of x and y
   float mass;             //given mass
   sf::Image image;        //image data
   sf::Texture texture;
   sf::Sprite sprite;
};
#endif
```

```
#include "CelestialBody.h"
//constructors
CelestialBody::CelestialBody() : xPos(0), yPos(0), xVel(0), yVel(0),
mass(0) {}
CelestialBody::CelestialBody(float x_position, float y_position, float
x_velocity, float y_velocity, float init_mass, std::string
image_filename) {
   if (!image.loadFromFile(image_filename)) {
       throw std::invalid_argument("Celestial Body image file not found,
exiting.");
   }
   texture.loadFromImage(image);
   sprite.setTexture(texture);
   xPos = x_position;   yPos = y_position;
   xVel = x_velocity;   yVel = y_velocity;
   mass = init_mass;
}
//accessors and mutators
float CelestialBody::getXPos() const { return xPos; }
float CelestialBody::getYPos() const { return yPos; }
float CelestialBody::getXVel() const { return xVel; }
float CelestialBody::getYVel() const { return yVel; }
float CelestialBody::getMass() const { return mass; }
sf::Image CelestialBody::getImage() const { return image; }
void CelestialBody::setXPos(float x_position) { xPos = x_position; }
void CelestialBody::setYPos(float y_position) { yPos = y_position; }
void CelestialBody::setXVel(float x_velocity) { xVel = x_velocity; }
void CelestialBody::setYVel(float y_velocity) { yVel = y_velocity; }
void CelestialBody::setMass(float mass) { mass = mass; }
void CelestialBody::setImage(std::string image_filename) {
   if(!image.loadFromFile(image_filename)) {
       throw std::invalid_argument("Celestial Body image file not
found, exiting.");
   }
   texture.loadFromImage(image);
   sprite.setTexture(texture);
}
//update x and y position of planet
void CelestialBody::updatePosition() {
   sprite.setPosition(xPos, yPos);
}
//override virtual draw function
void CelestialBody::draw(sf::RenderTarget& target, sf::RenderStates
states) const {
   target.draw(sprite, states);
}
```

```
//Thomas Freeman
//Implementation of Universe.h
//Header file declaring all methods in
//Universe.cpp
#ifndef UNIVERSE_H
#define UNIVERSE_H
#include "CelestialBody.h"
#include <vector>
class Universe : public sf::Drawable {
public:
   // Our constructors
   Universe();
   Universe(std::string image_filename);
   explicit Universe(const Universe& copyUniverse);
   explicit Universe(Universe&& moveUniverse) noexcept;
   ~Universe();
   //accessors and mutators
   int getNumOfPlanets() const;
   float getRadius() const;
   sf::Image getImage() const;
   void setNumOfPlanets(int n);
   void setRadius(float r);
   void setImage(std::string image_filename);
   //overloaded assignment operators
   Universe& operator=(Universe& rvalue);
   Universe& operator=(Universe&& rvalue) noexcept;
   //vector of pointers to celestial bodies
   std::vector<CelestialBody*> planets;
   //overloaded insertion operator for file input through terminal
   friend std::istream& operator>>(std::istream& in, Universe&
universe);

private:
  //override virtual draw function
   void draw(sf::RenderTarget& target, sf::RenderStates states) const
override;
   int numOfPlanets;
   float radius;               //radius of universe
   sf::Image image;            //image data for universe
   sf::Texture texture;
   sf::Sprite sprite;
};
#endif
```

```
//Thomas Freeman
//Implementation of Universe.cpp
//This program builds the universe from any
//of the Nbody files, throwing exceptions for
//incorrect inputs and gathering all data values
//needed to display the image
#include "Universe.h"
//Set of constructors to create the universe
Universe::Universe() : numOfPlanets(0), radius(0) {}
//Basic exception for unavailable image file
Universe::Universe(std::string image_filename) {
   if(!image.loadFromFile(image_filename)) {
       throw std::invalid_argument("Universe image file not found,
exiting.");
   }
   texture.loadFromImage(image);
   sprite.setTexture(texture);
   numOfPlanets = 0;
   radius = 0;
}
Universe::Universe(const Universe& copyUniverse) {
   numOfPlanets = copyUniverse.numOfPlanets;
       radius = copyUniverse.radius;
```

```
           image = copyUniverse.image;
               texture.loadFromImage(image);
                   sprite.setTexture(texture);
}
Universe::Universe(Universe&& moveUniverse) noexcept {
   numOfPlanets = moveUniverse.numOfPlanets;
       radius = moveUniverse.radius;
           image = moveUniverse.image;
               texture.loadFromImage(image);
                   sprite.setTexture(texture);
                       moveUniverse.numOfPlanets = 0;
                           moveUniverse.radius = 0;
}
Universe::~Universe() {
   numOfPlanets = 0;
   radius = 0;
}
//Allows the mutation and access of values
int Universe::getNumOfPlanets() const { return numOfPlanets; }
float Universe::getRadius() const { return radius; }
sf::Image Universe::getImage() const { return image; }
void Universe::setNumOfPlanets(int n) { numOfPlanets = n; }
void Universe::setRadius(float r) { radius = r; }
void Universe::setImage(std::string image_filename) {
   if(!image.loadFromFile(image_filename)) {
       throw std::invalid_argument("Universe image file not found,
exiting.");
   }
   texture.loadFromImage(image);
   sprite.setTexture(texture);
}
//overloaded assingment operators
Universe& Universe::operator=(Universe& rvalue) {
   if (this == &rvalue) { return *this; }
   numOfPlanets = rvalue.numOfPlanets;
       radius = rvalue.radius;
           image = rvalue.image;
               texture.loadFromImage(image);
                   sprite.setTexture(texture);
   return *this;
}
Universe& Universe::operator=(Universe&& rvalue) noexcept {
   if (this == &rvalue) { return *this; }
   numOfPlanets = rvalue.numOfPlanets;
       radius = rvalue.radius;
           image = rvalue.image;
               texture.loadFromImage(image);
                   sprite.setTexture(texture);
                       rvalue.numOfPlanets = 0;
                           rvalue.radius = 0;
   return *this;
}
// Overload the stream operator to allow terminal input
std::istream& operator>>(std::istream& in, Universe& universe) {
  float fval;
  std::string image;
  sf::Vector2u windowSize = universe.getImage().getSize();
  in >> universe.numOfPlanets;
  in >> universe.radius;
  // get the data for a planet
  for (int i = 0; i < universe.numOfPlanets; i++) {
      CelestialBody* planet = new CelestialBody();
      // Gets planet x
      in >> fval;
      fval = (((fval / 2) / universe.radius) * windowSize.x) +
(windowSize.x / 2);
       planet->setXPos(fval);
      //Gets planet y
      in >> fval;
      fval = (((fval / 2) / universe.radius) * windowSize.y) +
(windowSize.y / 2);
      planet->setYPos(fval);
      planet->updatePosition();
      //Gets necessary info
      in >> fval;
      planet->setXVel(fval);
      in >> fval;
      planet->setYVel(fval);
      in >> fval;
      planet->setMass(fval);
      in >> image;
      image = "nbody/" + image;
      try {
          planet->setImage(image);
      }
      catch (std::invalid_argument err) {
          std::cout << err.what() << std::endl;
          exit(-1);
      }
      universe.planets.push_back(planet);
  }

  return in;
}
//override virtual draw function
void Universe::draw(sf::RenderTarget& target, sf::RenderStates
states) const {
  target.draw(sprite, states);
}
```



Part two had us implement the correct physics and animation for our universe. The planets need their x and y position values updated every second based upon the x and y values of their velocity. In order to preserve the accuracy of the numbers in the text files, all calculations are done in double. These include Velocity, Acceleration, Force, Net force, and a special one, gravity. Gravity is special as it’s the only calculation based upon two celestial bodies. A function is made for pairwise force calculating on the pull between two bodies based upon their mass and distance from each other. As for extra credit, I added a loop into the main function that plays our given sound file and even created my own text file to simulate the universe based upon planets.txt. All that was needed was alternate values for xpos, ypos, xvel, etc, in order for the text file to update the positions when the program was ran.

The following is the code for ps2b

```
CC = g++
CFLAGS = -std=c++14 -c -g -Og -Wall -Werror -pedantic
OBJS = main.o CelestialBody.o Universe.o
LIBS = -lsfml-graphics -lsfml-audio -lsfml-window -lsfml-system
EXE = NBody
all: $(EXE)
NBody: $(OBJS)
	$(CC) $(OBJS) -o $(EXE) $(LIBS)
%.o: %.cpp
	$(CC) $(CFLAGS) -o $@ $<
clean:
	\rm $(OBJS) $(EXE)
```

```
//Thomas Freeman
//Implementation of main.cpp
//main deals with the command line,
//providing our necessary command and the data
//from our input
#include "Universe.h"
#include <fstream>
#include <SFML/Audio.hpp>
// This time our command takes
int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cerr << "usage: ./NBody T Î”t < input_file" << std::endl;
        exit(-1);
    }
    double time = strtod(argv[1], nullptr);
    double deltaT = strtod(argv[2], nullptr);
    int elapsed_time = 0;
    Universe universe;
    try {
        universe = Universe("nbody/starfield.jpg");
        // Gets universe data from input file
        std::cin >> universe;
    }
    catch (const std::invalid_argument& err) {
        std::cout << err.what() << std::endl;
        exit(-1);
    }
    sf::Vector2u windowSize = universe.getImage().getSize();
    sf::RenderWindow window(sf::VideoMode(windowSize.x, windowSize.y),
    "The Solar System!");
    window.setFramerateLimit(60);
  // Plays sound file from nbody
    sf::Music music;
    if (!music.openFromFile("nbody/2001.wav")) {
        std::cout << "Cannot load music nbody/2001.wav" << std::endl;
    }
    music.play();
    music.setLoop(true);
    sf::Text timer;
    sf::Font font;
    if (!font.loadFromFile("nbody/verdana.ttf")) {
      std::cout << "Cannot load font nbody/verdana.ttf" << std::endl;
    }
    timer.setFont(font);
    timer.setFillColor(sf::Color::White);
    timer.setCharacterSize(16);
    // SFML display loop
    while (window.isOpen()) {
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                window.close();
            }
        }
        timer.setString("Elapsed time: " + std::to_string(elapsed_time)+
        "seconds");
        window.clear();
        window.draw(universe);
        for (int i = 0; i < universe.getNumOfPlanets(); i++) {
            window.draw(*universe.planets[i]);
        }
        window.draw(timer);
        if (elapsed_time < time) {
            universe.step(deltaT);      // call to step function
            elapsed_time += deltaT;
        }
        else music.stop();
        window.display();
    }
    // prints final positions of planets to output.txt
    std::ofstream outfile;
    outfile.open("output.txt");
    outfile << universe.getNumOfPlanets() << std::endl;
    outfile << std::scientific << universe.getRadius() << std::endl;
    for (int i = 0; i < universe.getNumOfPlanets(); i++) {
        outfile << std::scientific << universe.planets[i]->getXPos()
        << "  " << std::scientific << universe.planets[i]->getYPos()
        << "  " << std::scientific << universe.planets[i]->getXVel()
        << "  " << std::scientific << universe.planets[i]->getYVel()
        << "  " << std::scientific << universe.planets[i]->getMass()
        << "  " << universe.planets[i]->getImageFilename() << std::endl;
    }
    outfile.close();
    return 0;
}
```

```
//Thomas Freeman
//implementation of CelestialBody.h
//header file defining all methods a variables present in
//CelestialBody.cpp
#ifndef CELESTIALBODY_H
#define CELESTIALBODY_H
#include <iostream>
#include <exception>
#include <SFML/Graphics.hpp>
class CelestialBody : public sf::Drawable {
public:
    // constructors
    CelestialBody();
    CelestialBody(double x_position, double y_position, double
x_velocity,
    double y_velocity, double init_mass, std::string image_filename);
    // accessors and mutators
```

```
   double getXPos() const;
   double getYPos() const;
   double getXVel() const;
   double getYVel() const;
   double getMass() const;
   sf::Image getImage() const;
   std::string getImageFilename() const;
   void setXPos(double x_position);
   void setYPos(double y_position);
   void setXVel(double x_velocity);
   void setYVel(double y_velocity);
   void setMass(double _mass);
   void setImage(std::string image_filename);
   // update x and y position of planet given the scale of the universe
   void updatePosition(double scale, sf::Vector2u windowSize);
private:
   // override virtual draw function
   void draw(sf::RenderTarget& target, sf::RenderStates states) const
   override;
   double xPos, yPos;      // x and y position
   double xVel, yVel;      // x and y velocity
   double mass;
   std::string imageFilename;
   sf::Image image;
   sf::Texture texture;
   sf::Sprite sprite;
};
#endif
```

```
#include "CelestialBody.h"
// constructors
CelestialBody::CelestialBody() : xPos(0), yPos(0), xVel(0), yVel(0),
mass(0) {}
CelestialBody::CelestialBody(double x_position, double y_position,
double x_velocity, double y_velocity, double init_mass,
std::string image_filename) {
   if (!image.loadFromFile(image_filename)) {
      throw std::invalid_argument("Celestial Body image file not found,"
       " exiting");
   }
   imageFilename = image_filename;
   texture.loadFromImage(image);
   sprite.setTexture(texture);
   xPos = x_position;   yPos = y_position;
   xVel = x_velocity;   yVel = y_velocity;
   mass = init_mass;
}
// accessors and mutators
double CelestialBody::getXPos() const { return xPos; }
double CelestialBody::getYPos() const { return yPos; }
double CelestialBody::getXVel() const { return xVel; }
double CelestialBody::getYVel() const { return yVel; }
double CelestialBody::getMass() const { return mass; }
sf::Image CelestialBody::getImage() const { return image; }
std::string CelestialBody::getImageFilename() const {
    return imageFilename;
}
void CelestialBody::setXPos(double x_position) { xPos = x_position; }
void CelestialBody::setYPos(double y_position) { yPos = y_position; }
void CelestialBody::setXVel(double x_velocity) { xVel = x_velocity; }
void CelestialBody::setYVel(double y_velocity) { yVel = y_velocity; }
void CelestialBody::setMass(double _mass) { mass = _mass; }
void CelestialBody::setImage(std::string image_filename) {
    if(!image.loadFromFile("nbody/" + image_filename)) {
        throw std::invalid_argument("Celestial Body image file not
found");
    }
    imageFilename = image_filename;
    texture.loadFromImage(image);
    sprite.setTexture(texture);
}
//update x and y position of planet
void CelestialBody::updatePosition(double scale, sf::Vector2u
windowSize) {
    int x, y;
    x = ((xPos / 2) / scale * windowSize.x) + (windowSize.x / 2);
    y = ((yPos / 2) / scale * windowSize.y) + (windowSize.y / 2);
    sprite.setPosition(x, y);
}
//override virtual draw function
void CelestialBody::draw(sf::RenderTarget& target, sf::RenderStates
states)
const {
     target.draw(sprite, states);
}
```

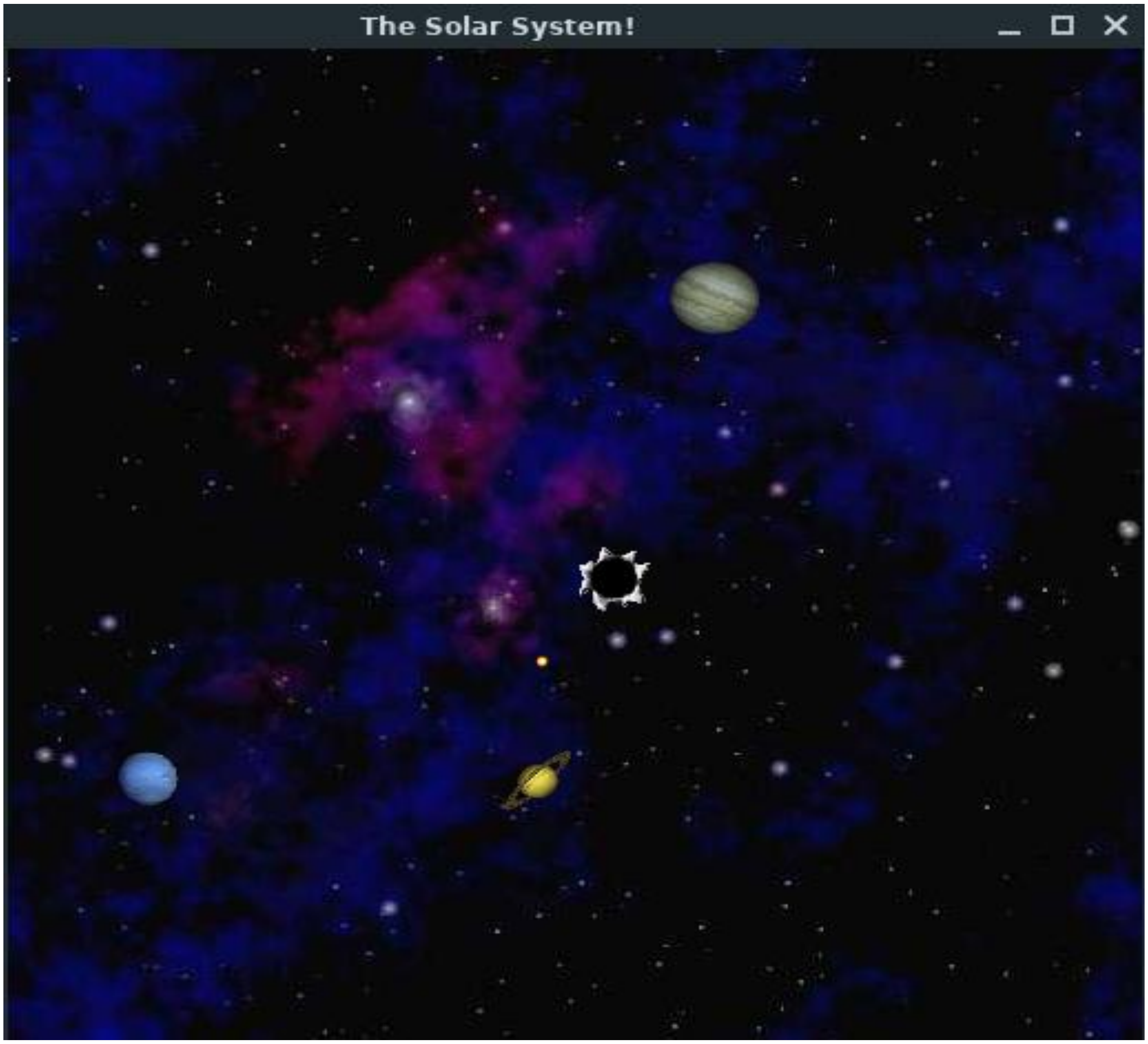
```
//Thomas Freeman
//Implementation of Universe.h
//Header file declaring all methods in
//Universe.cpp
#ifndef UNIVERSE_H
#define UNIVERSE_H
#include "CelestialBody.h"
#include <memory>
#include <vector>
#include <cmath>
#define GRAVITY_CONST 6.67e-11
class Universe : public sf::Drawable {
public:
    // constructors
    Universe();
    Universe(std::string image_filename);
    explicit Universe(const Universe& copyUniverse);
    explicit Universe(Universe&& moveUniverse) noexcept;
    ~Universe();
    // accessors and mutators
    int getNumOfPlanets() const;
    double getRadius() const;
    sf::Image getImage() const;
    void setNumOfPlanets(int n);
    void setRadius(double r);
    void setImage(std::string image_filename);
    /* step moves forward the simulation a single interval of time in
    seconds */
    void step(double seconds);
    // vector of pointers to celestial bodies
    std::vector<std::unique_ptr<CelestialBody>> planets;
    // overloaded assignment operators for Universe setup in main
    Universe& operator=(Universe& rvalue);
    Universe& operator=(Universe&& rvalue) noexcept;
    // overloaded insertion operator for file input through terminal
    friend std::istream& operator>>(std::istream& in, Universe&
universe);

private:
    // override virtual draw function
    void draw(sf::RenderTarget& target, sf::RenderStates states) const
    override;
    int numOfPlanets;
    double radius;              // radius of the universe
    sf::Image image;            // universe background
    sf::Texture texture;
    sf::Sprite sprite;
};
#endif
```

```
//Thomas Freeman
//Implementation of Universe.cpp
//This program builds the universe from any
//of the Nbody files, throwing exceptions for
//incorrect inputs and gathering all data values
//needed to display the image
#include "Universe.h"
//Set of constructors to create the universe
Universe::Universe() : numOfPlanets(0), radius(0) {}
//Basic exception for unavailable image file
Universe::Universe(std::string image_filename) {
    if(!image.loadFromFile(image_filename)) {
        throw std::invalid_argument("Universe image file not found,"
        " exiting");
    }
    texture.loadFromImage(image);
    sprite.setTexture(texture);
```

```
   numOfPlanets = 0;
   radius = 0;
}
Universe::Universe(const Universe& copyUniverse) {
   numOfPlanets = copyUniverse.numOfPlanets;
       radius = copyUniverse.radius;
           image = copyUniverse.image;
               texture.loadFromImage(image);
                   sprite.setTexture(texture);
}
Universe::Universe(Universe&& moveUniverse) noexcept {
   numOfPlanets = moveUniverse.numOfPlanets;
       radius = moveUniverse.radius;
           image = moveUniverse.image;
               texture.loadFromImage(image);
                   sprite.setTexture(texture);
                       moveUniverse.numOfPlanets = 0;
                           moveUniverse.radius = 0;
}
Universe::~Universe() {
   numOfPlanets = 0;
   radius = 0;
}
// accessors and mutators
int Universe::getNumOfPlanets() const { return numOfPlanets; }
double Universe::getRadius() const { return radius; }
sf::Image Universe::getImage() const { return image; }
void Universe::setNumOfPlanets(int n) { numOfPlanets = n; }
void Universe::setRadius(double r) { radius = r; }
void Universe::setImage(std::string image_filename) {
   if(!image.loadFromFile(image_filename)) {
       throw std::invalid_argument("Universe image file not found");
   }
   texture.loadFromImage(image);
   sprite.setTexture(texture);
}
// Moves the simulation forward one second, altering delta x and delta
y every second
void Universe::step(double seconds) {
   double d, deltaX, deltaY;       // distance d formula
   double force, forceX, forceY;   // force calculation
   double accelX, accelY;          // acceleration calculation
   for (int i = 0; i < numOfPlanets; i++) {
       forceX = 0; forceY = 0;
       // Calculates the Net force on x and y
       for (int j = 0; j < numOfPlanets; j++) {
           if (i != j) {
               deltaX = planets[j]->getXPos() - planets[i]->getXPos();
               deltaY = planets[j]->getYPos() - planets[i]->getYPos();
               d = sqrt(deltaX * deltaX + deltaY * deltaY);
               //Calculates the pairwise force, a product of
gravitational constants
               //and the mass of two planets divided by the square of
the distance between them
```

```
               force = (GRAVITY_CONST * planets[i]->getMass() *
               planets[j]->getMass()) / (d * d);
               forceX += force * (deltaX / d);
               forceY += force * (deltaY / d);
           }
       }
       // Acceleration calculation
       accelX = forceX / planets[i]->getMass();
       accelY = forceY / planets[i]->getMass();
       // Velocity calculation
       planets[i]->setXVel(planets[i]->getXVel() + seconds * accelX);
       planets[i]->setYVel(planets[i]->getYVel() + seconds * accelY);
   }
   // Positions are updated
   for (int i = 0; i < numOfPlanets; i++) {
       planets[i]->setXPos(planets[i]->getXPos() + seconds *
       planets[i]->getXVel());
       planets[i]->setYPos(planets[i]->getYPos() + seconds *
       planets[i]->getYVel());
       planets[i]->updatePosition(radius, image.getSize());
   }
}
// overloaded assingment operators
Universe& Universe::operator=(Universe& rvalue) {
   if (this == &rvalue) { return *this; }
   numOfPlanets = rvalue.numOfPlanets;
       radius = rvalue.radius;
           image = rvalue.image;
               texture.loadFromImage(image);
                   sprite.setTexture(texture);
   return *this;
}
Universe& Universe::operator=(Universe&& rvalue) noexcept {
   if (this == &rvalue) { return *this; }
   numOfPlanets = rvalue.numOfPlanets;
       radius = rvalue.radius;
           image = rvalue.image;
               texture.loadFromImage(image);
                   sprite.setTexture(texture);
                       rvalue.numOfPlanets = 0;
                           rvalue.radius = 0;
   return *this;
}
// overload stream operator for command input
Std::istream& operator>>(std::istream& in, Universe& universe) {
   double dval;
   char buffer[30];    // Files should be less than 30 characters
   std::string image;
   in >> universe.numOfPlanets;
   if (universe.numOfPlanets < 0) { universe.numOfPlanets = 0; }
   else if (universe.numOfPlanets > 1000) {
       universe.numOfPlanets = 1000;
   }
   in >> universe.radius;
```

```
    // finds each planets data
    for (int i = 0; i < universe.numOfPlanets; i++) {
        std::unique_ptr<CelestialBody> planet(
            std::make_unique<CelestialBody>(CelestialBody()));
        //gets all necessary data from the txt file
        in >> dval;
        planet->setXPos(dval);
        in >> dval;
        planet->setYPos(dval);
        planet->updatePosition(universe.radius,
universe.image.getSize());
        in >> dval;
        planet->setXVel(dval);
        in >> dval;
        planet->setYVel(dval);
        in >> dval;
        planet->setMass(dval);
        in >> buffer;
        image = buffer;

        try {
            planet->setImage(image);
        }
        catch (const std::invalid_argument& err) {
            std::cout << err.what() << std::endl;
            exit(-1);
        }
        universe.planets.push_back(std::move(planet));
    }
    return in;
}
// override virtual draw function
void Universe::draw(sf::RenderTarget& target, sf::RenderStates states)
const {
    target.draw(sprite, states);
}
```



PS3 – Recursive Graphics

The goal for assignment 3 was to implement the Sierpinski algorithm. An algorithm that is used to generate triangles in a specific pattern around the base triangle. This assignment was my first experience using CPPlint, a python application that is used to analyze our code we write and give us suggestions on how to rewrite our programs, such that they satisfy googles coding style guidelines. We implemented this in a recursive fashion as creating the algorithm iteratively results in a slow and bugged mess. Every recursive call increases the depth triangles by one, drawing 3 triangles around each one. How are the other triangles drawn? For my implementation I used vectors to ensure that for each side of the triangle, another triangle would be drawn for all depths. The function Convex Shape also proved helpful as it determines the order in which the triangles are drawn for a give depth. The program is effective up until the number of calls becomes too much to process. In my experience I noticed that after 13 recursive calls, there isn't enough memory to process the remaining calls, resulting in a crash of the virtual machine. The Example below shows the program running with a triangle length of six and algorithmic depth of eight.

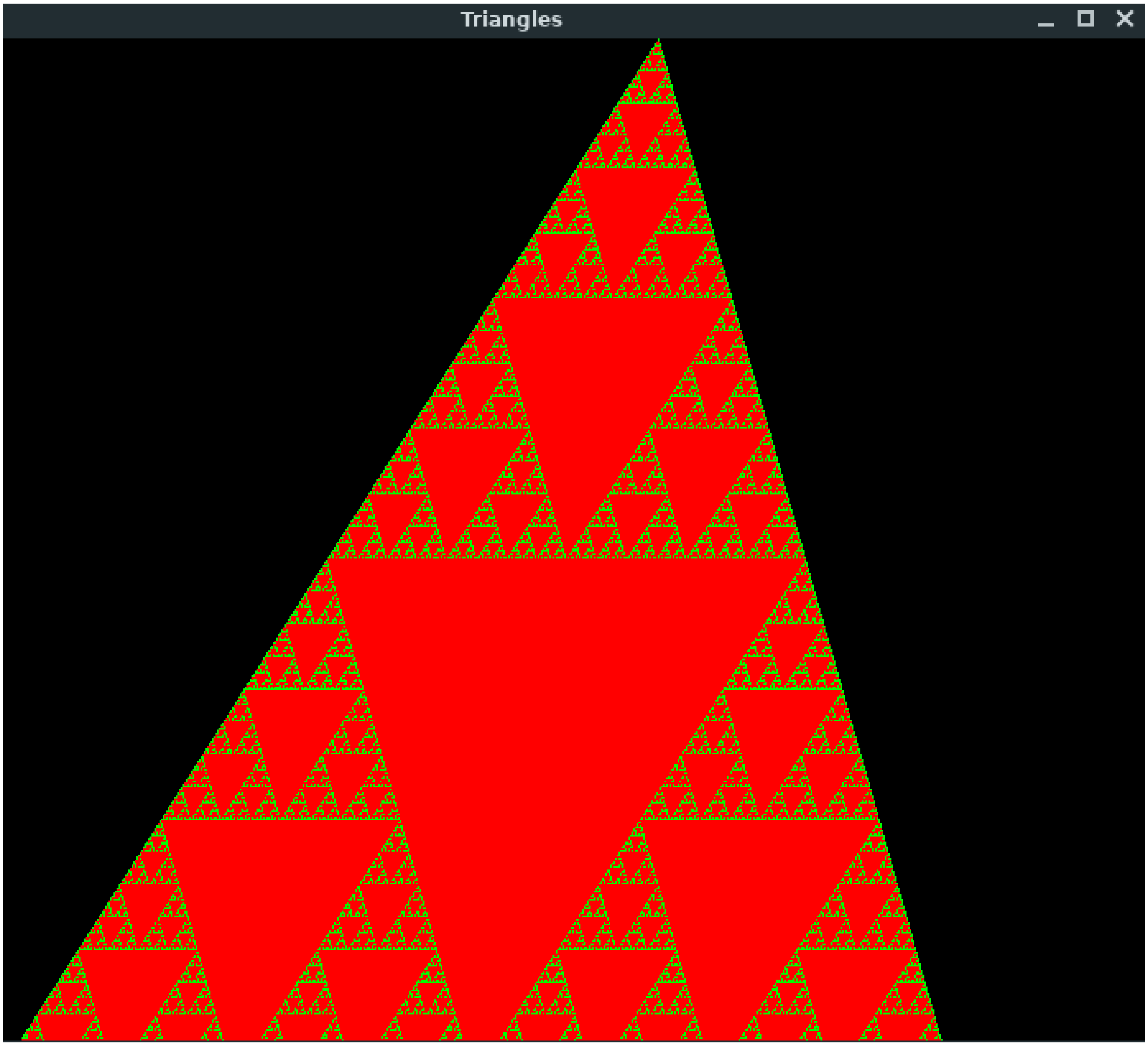
The following files are associated with PS3

```
CC = g++
CFLAGS = -Wall -Werror -ansi -pedantic
LFLAGS = -lsfml-graphics -lsfml-window -lsfml-system
all: Triangle
Triangle: Tfractal.o Triangle.o
    $(CC) Tfractal.o Triangle.o -o Triangle $(LFLAGS)
Tfractal.o:    Tfractal.cpp Triangle.hpp
    $(CC) -c Tfractal.cpp $(CFLAGS)
Triangle.o: Triangle.cpp Triangle.hpp
    $(CC) -c Triangle.cpp $(CFLAGS)
clean:
    rm Triangle
    rm Tfractal.o
    rm Triangle.o
```

```
// copyright 2021 Thomas Freeman
#include "Triangle.hpp"
int main(int argc, char* argv[]) {
 if (argc < 3 || argc > 4) {
   cout << "./Triangle [L] [N]" << endl;
   return -1;
 }
 int side = atoi(argv[1]);
 int depth = atoi(argv[2]);
 if (depth < 0) {
   cout << "depth should be greater than 0" << endl;
   return -1;
 }
 Triangle obj(side, depth);
 int window_height = 0.5 * sqrt(3.0) * side;
 RenderWindow window(VideoMode(side, window_height), "Triangles");
 window.setFramerateLimit(1);
 while (window.isOpen()) {
   Event event;
   while (window.pollEvent(event))
     if (event.type == Event::Closed)
       window.close();
   window.clear();
   window.draw(obj);
   window.display();
 }
 return 0;
}  // end of Tfractal

```

```
// copyright 2021 Thomas Freeman
#ifndef TRIANGLE_HPP  //NOLINT
#define TRIANGLE_HPP  //NOLINT
#include <cmath>
#include <iostream>
#include <vector>
#include <SFML/Graphics.hpp>
using std::cout;
using std::endl;
using sf::Color;
using sf::Vector2f;
using sf::RenderTarget;
using sf::RenderWindow;
using sf::RenderStates;
using sf::VideoMode;
using sf::Event;
using sf::ConvexShape;
using sf::Drawable;
class Triangle :  public Drawable {
public:
 Triangle(int, int);
 Triangle(Vector2f, Vector2f, Vector2f, int);
 void virtual draw(RenderTarget&, RenderStates) const;
private:
 int ftree;
 Vector2f t, l, r;
 Vector2f p1, p2, p3;
 Triangle *t1, *t2, *t3;
};
#endif //NOLINT
```

```
// copyright 2021 Thomas Freeman
#include "Triangle.hpp"
Triangle::Triangle(int side, int depth) : ftree(depth) {
 t = Vector2f(side / sqrt(3.0), 0);
 float height = .5 * sqrt(3.0) * side;
 l = Vector2f(0, height);
 r = Vector2f(side-1, height);
 p1 = Vector2f(((t.x + l.x) / 2), ((t.y + l.y) / 2) );
 p2 = Vector2f(((l.x + r.x) / 2), ((l.y + r.y) / 2) );
 p3 = Vector2f(((t.x + r.x) / 2), ((t.y + r.y) / 2) );
 if (ftree - 1 > 0) {
   t1 = new Triangle(p1, l, p2, depth - 1);
   t2 = new Triangle(t, p1, p3, depth - 1);
   t3 = new Triangle(p3, p2, r, depth - 1);
 } else {
   t1 = NULL;
   t2 = NULL;
   t3 = NULL;
 }
}
Triangle::Triangle(Vector2f top, Vector2f left,
Vector2f right, int depth) : ftree(depth) {
 if (depth <= 0) {
   t1 = NULL;
   t2 = NULL;
   t3 = NULL;
   return;
 }
 t = top;
 l = left;
 r = right;
 p1 = Vector2f(((top.x + left.x) / 2), ((top.y + left.y) / 2));
 p2 = Vector2f(((left.x + right.x) / 2), ((left.y + right.y) / 2));
 p3 = Vector2f(((top.x + right.x) / 2), ((top.y + right.y) / 2));
 t1 = new Triangle(p1, left, p2, depth - 1);
 t2 = new Triangle(top, p1, p3, depth - 1);
 t3 = new Triangle(p3, p2, right, depth - 1);
}
void Triangle::draw(RenderTarget& target, RenderStates states) const {
 ConvexShape triangle;
 triangle.setPointCount(3);
 triangle.setPoint(0, l);
 triangle.setPoint(1, r);
 triangle.setPoint(2, t);
 triangle.setFillColor(Color::Green);
 ConvexShape triangle1;
 triangle1.setPointCount(3);
 triangle1.setPoint(0, p1);
 triangle1.setPoint(1, p2);
 triangle1.setPoint(2, p3);
 triangle1.setFillColor(Color::Red);
 target.draw(triangle);
 if (ftree > 0)
   target.draw(triangle1);
 if (t1 != NULL) {
   t1->draw(target, states);
   t2->draw(target, states);
   t3->draw(target, states);
 }
}
```



PS4a – Synthesizing a Plucked String sound

This is assignment is more of the setup for PS4b, as the main goal was to create a ring buffer, A data structure that can simulate a string that energy travels through, simulating sound in the final implementation. The idea of a ring buffer comes from the Karplus Strong algorithm which simulates sound based upon different designs of feedback loops. A ring buffer happens to be the one implemented. The Ring Buffer operates like a dynamic Array using the two functions enqueue and dequeue, adding data to the stacks end and popping data from the top.

CircularBuffer also copies and destructs data constantly as enqueue and dequeue run their course through the ring buffer. Since We need at most one loop this is very efficient with a time complexity of 0(N).

The following is the Code for PS4a

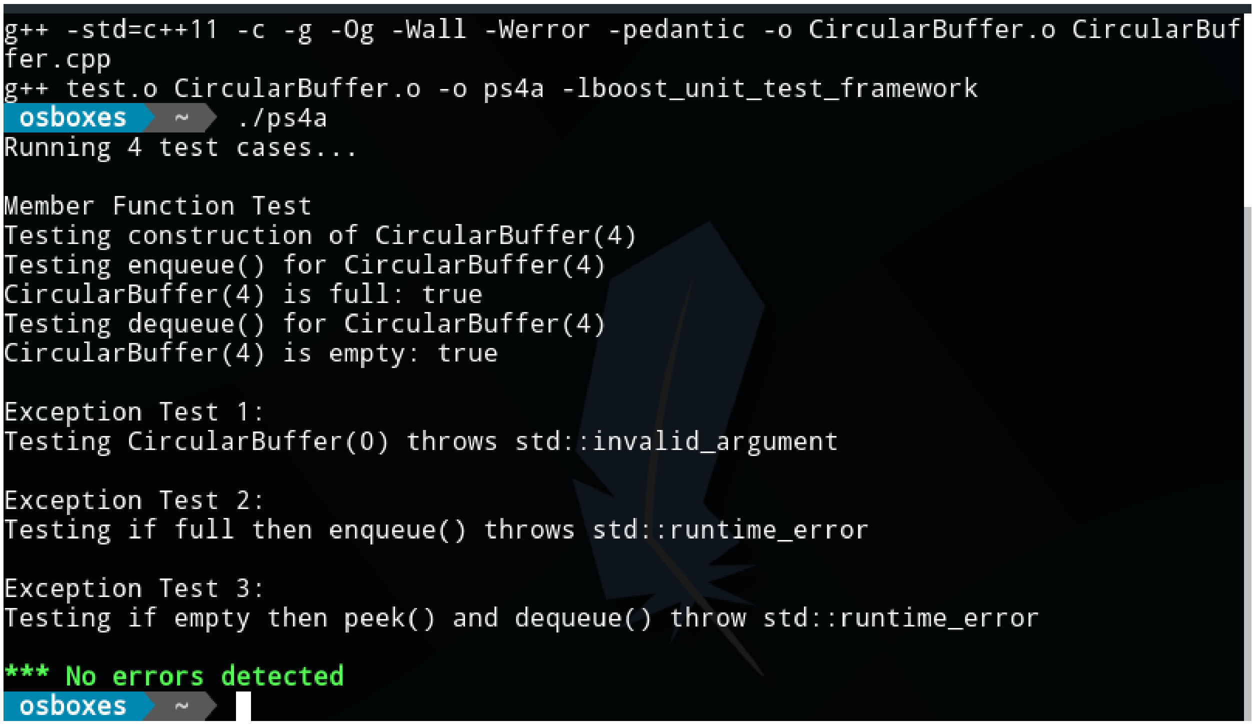
```
CC = g++
CFLAGS = -std=c++11 -c -g -Og -Wall -Werror -pedantic
LIBS = -lboost_unit_test_framework
OBJS = test.o CircularBuffer.o
EXE = ps4a
all : $(EXE)
$(EXE) : $(OBJS)
	$(CC) $(OBJS) -o $(EXE) $(LIBS)
%.o : %.cpp
	$(CC) $(CFLAGS) -o $@ $<
clean :
	\rm $(OBJS) $(EXE)
```

```
// copyright 2021 Thomas Freeman
#ifndef CIRCULARBUFFER_H   //NOLINT
#define CIRCULARBUFFER_H   //NOLINT
#include <stdint.h>
#include <iostream>
#include <exception>
class CircularBuffer {
public:
explicit CircularBuffer(int capacity);
// ring buffer is made empty
// with its own max capacity
CircularBuffer(const CircularBuffer& copyCB);
CircularBuffer(CircularBuffer&& moveCB) noexcept;
~CircularBuffer();
int size() const;          // return the elements in the buffer
bool isEmpty() const;      // checks if size = 0
bool isFull() const;       // checks if size = capacity
void enqueue(int16_t x);   // add an item
int16_t dequeue();         // deletes an item
int16_t peek();            // brings an item to the front of the buffer
void empty();
CircularBuffer& operator=(const CircularBuffer& rightSide);
CircularBuffer& operator=(CircularBuffer&& rvalue) noexcept;
private:
int _size;
int _capacity;
int _first;
int _last;
int16_t* _buffer;
};
#endif  //NOLINT
```

```
// copyright 2021 Thomas Freeman
#include "CircularBuffer.hpp"
// List of constructors
CircularBuffer::CircularBuffer(int capacity) {
    if (capacity < 1) {
        throw std::invalid_argument(" Capacity must be be greater than
zero.");
    }
    _capacity = capacity;
    _size = 0;
    _first = 0;
    _last = 0;
    _buffer = new int16_t[capacity];
}
CircularBuffer::CircularBuffer(const CircularBuffer& copyCB) {
    _capacity = copyCB._capacity;
    if (_capacity > 0) {
        _size = copyCB._size;
        _first = copyCB._first;
        _last = copyCB._last;
        _buffer = new int16_t[_capacity];
        for (int i = 0; i < _size; i++) {
            _buffer[i] = copyCB._buffer[i];
        }
    } else {
        _size = 0;
        _first = 0;
        _last = 0;
        _buffer = nullptr;
    }
}
CircularBuffer::CircularBuffer(CircularBuffer&& moveCB) noexcept {
    _capacity = moveCB._capacity;
    _size = moveCB._size;
    _first = moveCB._first;
    _last = moveCB._last;
    _buffer = moveCB._buffer;
    moveCB._capacity = 0;
    moveCB._size = 0;
    moveCB._first = 0;
    moveCB._last = 0;
    moveCB._buffer = nullptr;
}
CircularBuffer::~CircularBuffer() {
    if (_buffer != nullptr) { delete[] _buffer; }
    _capacity = 0;
    _size = 0;
    _first = 0;
    _last = 0;
    _buffer = nullptr;
}
// main functions in circular buffer
int CircularBuffer::size() const { return _size; }
bool CircularBuffer::isEmpty() const { return _size == 0; }
bool CircularBuffer::isFull() const { return _size == _capacity; }
void CircularBuffer::enqueue(int16_t x) {
    if (isFull()) {
        throw std::runtime_error("enqueue: can't enqueue to a full
ring.");
    }
    // we add an item to the buffer then increment
    _buffer[_last] = x;
    _last++;
    if (_last == _capacity) { _last = 0; }
    _size++;
}
int16_t CircularBuffer::dequeue() {
    if (isEmpty()) {
        throw std::runtime_error("dequeue: can't dequeue from an empty
ring.");
    }
    // return the first item then index the new first item
    int16_t result = _buffer[_first];
    _first++;
    if (_first == _capacity) { _first = 0; }
    _size--;
    return result;
}
// peek exception runs, returns first item after
int16_t CircularBuffer::peek() {
    if (isEmpty()) {
        throw std::runtime_error("peek: can't peek from an empty
ring.");
    }
    return _buffer[_first];
}
void CircularBuffer::empty() {
    _size = 0;
    _first = 0;
    _last = 0;
}
/** overloaded assignment operators **/
CircularBuffer& CircularBuffer::operator=(const CircularBuffer&
rightSide) {
    if (this == &rightSide) { return *this; }
    if (_buffer != nullptr) { delete[] _buffer; }
    _capacity = rightSide._capacity;
    if (_capacity > 0) {
        _size = rightSide._size;
        _first = rightSide._first;
        _last = rightSide._last;
        _buffer = new int16_t[_capacity];
        for (int i = 0; i < _size; i++) {
            _buffer[i] = rightSide._buffer[i];
        }
    } else {
        _size = 0;
        _first = 0;
        _last = 0;
        _buffer = nullptr;
    }
    return *this;
}
CircularBuffer& CircularBuffer::operator=(CircularBuffer&& rvalue)
noexcept {
    if (this == &rvalue) { return *this; }
    if (_buffer != nullptr) { delete[] _buffer; }
    _capacity = rvalue._capacity;
    _size = rvalue._size;
    _first = rvalue._first;
    _last = rvalue._last;
    _buffer = rvalue._buffer;
    rvalue._capacity = 0;
    rvalue._size = 0;
    rvalue._first = 0;
    rvalue._last = 0;
    rvalue._buffer = nullptr;
    return *this;
}
```

```
// Copyright 2021 Thomas Freeman
#include "CircularBuffer.hpp"
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Main
#include <boost/test/unit_test.hpp>
// Initial test for Circular Buffer
BOOST_AUTO_TEST_CASE(functions_test) {
    std::cout << "\nMember Function Test " << std::endl;
    std::cout << "Testing construction of CircularBuffer(4)" <<
std::endl;
    BOOST_REQUIRE_NO_THROW(CircularBuffer(4));
    // Enqueue testing
    std::cout << "Testing enqueue() for CircularBuffer(4)" << std::endl;
    CircularBuffer cb(4);
    BOOST_REQUIRE_NO_THROW(cb.enqueue(1));
    BOOST_REQUIRE_NO_THROW(cb.enqueue(2));
    BOOST_REQUIRE_NO_THROW(cb.enqueue(3));
    BOOST_REQUIRE_NO_THROW(cb.enqueue(4));
    std::cout << "CircularBuffer(4) is full: " << std::boolalpha <<
    cb.isFull() << std::endl;
    // Dequeue testing
    std::cout << "Testing dequeue() for CircularBuffer(4)" << std::endl;
    BOOST_REQUIRE_NO_THROW(cb.dequeue());
    BOOST_REQUIRE_NO_THROW(cb.dequeue());
    BOOST_REQUIRE_NO_THROW(cb.dequeue());
```

```
   BOOST_REQUIRE_NO_THROW(cb.dequeue());
   std::cout << "CircularBuffer(4) is empty: " << std::boolalpha <<
   cb.isEmpty() << std::endl << std::endl;
}
// Exception test for capacity
BOOST_AUTO_TEST_CASE(excpetion_test1) {
  std::cout << "Exception Test 1: " <<
  std::endl << "Testing CircularBuffer(0) throws
std::invalid_argument" <<
   std::endl << std::endl;
   BOOST_REQUIRE_THROW(CircularBuffer cb1(0), std::invalid_argument);
}
// Enqueue test exception for adding the full buffer
BOOST_AUTO_TEST_CASE(excpetion_test2) {
   std::cout << "Exception Test 2: " <<
   std::endl << "Testing if full then enqueue() throws
std::runtime_error"
   << std::endl << std::endl;
   CircularBuffer cb2(4);
   cb2.enqueue(1);
   cb2.enqueue(2);
   cb2.enqueue(3);
   cb2.enqueue(4);
   BOOST_REQUIRE_THROW(cb2.enqueue(5), std::runtime_error);
}
// Tests the dequeue exception for dequeueing an empty buffer.
BOOST_AUTO_TEST_CASE(exception_test3) {
   std::cout << "Exception Test 3: " <<
   std::endl << "Testing if empty then peek() and dequeue() throw "
   "std::runtime_error" << std::endl;
   CircularBuffer cb3(1);
   BOOST_REQUIRE_THROW(cb3.peek(), std::runtime_error);
   BOOST_REQUIRE_THROW(cb3.dequeue(), std::runtime_error);
}

```



```
g++ -std=c++11 -c -g -Og -Wall -Werror -pedantic -o CircularBuffer.o CircularBuf
fer.cpp
g++ test.o CircularBuffer.o -o ps4a -lboost_unit_test_framework
 osboxes  ~  ./ps4a
Running 4 test cases...

Member Function Test
Testing construction of CircularBuffer(4)
Testing enqueue() for CircularBuffer(4)
CircularBuffer(4) is full: true
Testing dequeue() for CircularBuffer(4)
CircularBuffer(4) is empty: true

Exception Test 1:
Testing CircularBuffer(0) throws std::invalid_argument

Exception Test 2:
Testing if full then enqueue() throws std::runtime_error

Exception Test 3:
Testing if empty then peek() and dequeue() throw std::runtime_error

*** No errors detected
 osboxes  ~ 
```

PS4b – String Sound Implementation and SFML Audio Output

Now we get to play some music by implementing our next class, String Sound. String Sound Is a list of constructors that can extract the values generated by our pluck methods Random number generator. These values are then enqueued into Int_16t as samples. We then create KSGuitarSim which is the main method that gives us the key binds and picture of a keyboard in order to play the Guitar Strings from lowest to highest pitch. The samples are created via the Frequency which comes from the String Sound class. Pluck is assigned a time for each pressed key. There are also exceptions included for Unloadable sounds from the buffer.

The following is the code for PS4b

```
CC = g++
CFLAGS = -std=c++11 -c -g -Og
LIBS = -lsfml-graphics -lsfml-audio -lsfml-window -lsfml-system
OBJS = KSGuitarSim.o StringSound.o CircularBuffer.o
EXE = KSGuitarSim
all : $(EXE)
$(EXE) : $(OBJS)
    $(CC) $(OBJS) -o $(EXE) $(LIBS)
%.o : %.cpp
    $(CC) $(CFLAGS) -o $@ $<
clean :
    \rm $(OBJS) $(EXE)
```

```
// Copyright 2021 Thomas Freeman
```

```
#include <SFML/Graphics.hpp>
#include <SFML/System.hpp>
#include <SFML/Audio.hpp>
#include <SFML/Window.hpp>
#include "StringSound.hpp"
// takes input from string sound to make vector of samples
std::vector<sf::Int16> makeSamples(StringSound* gs);
// initializes said vector
void setupSoundBuffers(const std::vector<std::vector<sf::Int16>>&
samples,
std::vector<sf::SoundBuffer>* soundBuffers);
// initializes vector from sound buffers
void setupSounds(const std::vector<sf::SoundBuffer>& soundBuffers,
std::vector<sf::Sound>* sounds);
// Display setup
int main(int argc, char* argv[]) {
    sf::Image image;
    sf::Texture texture;
    sf::Sprite sprite;
    if (!image.loadFromFile("Keys.png")) {
        throw std::runtime_error("sf::Image: could not load Keys.png");
    }
    texture.loadFromImage(image);
    sprite.setTexture(texture);
    sf::RenderWindow window(sf::VideoMode(image.getSize().x,
    image.getSize().y), "KS Guitar Sim");
    // keys used to play guitar
    std::string keys = "q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,.;/'";
    std::vector<std::vector<sf::Int16>> samples;
    std::vector<sf::SoundBuffer> soundBuffers;
    std::vector<sf::Sound> sounds;
    StringSound gs;
    // Frequency determines sound
    double freq;
    // sampes are created for each key
    for (int i = 0; i < static_cast<int>(keys.length()); i++) {
        freq = 440 * pow(2, (static_cast<double>(i) - 24) / 12);
        try {
            gs = StringSound(freq);
        }
        catch (std::invalid_argument err) {
            std::cerr << err.what() << std::endl;
        }
        samples.push_back(makeSamples(&gs));
    }
    setupSoundBuffers(samples, &soundBuffers);
    setupSounds(soundBuffers, &sounds);
    while (window.isOpen()) {
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                window.close();
            }
    // Checks for pressed key, if pressed, plays given sound sample
```

```
           if (event.type == sf::Event::TextEntered) {
               char key = static_cast<char>(event.text.unicode);
               for (int i = 0; i < static_cast<int>(keys.length());
i++) {
                   if (key == keys[i]) {
                       sounds[i].play();
                   }
               }
           }
       }
       window.clear();
       window.draw(sprite);
       window.display();
   }
   return 0;
}

// includes the duration of pluck as int
std::vector<sf::Int16> makeSamples(StringSound* gs) {
   std::vector<sf::Int16> sampleStream;
   gs->pluck();
   int duration = 8;
   for (int i = 0; i < SAMPLE_RATE * duration; i++) {
       gs->tic();
       sampleStream.push_back(gs->sample());
   }
   return sampleStream;
}
// Exception for unloadble sound buffer
void setupSoundBuffers(const std::vector<std::vector<sf::Int16>>&
samples,
std::vector<sf::SoundBuffer>* soundBuffers) {
   sf::SoundBuffer buffer;
   for (int i = 0; i < static_cast<int>(samples.size()); i++) {
       if (!buffer.loadFromSamples(&(samples[i])[0], samples[i].size(),
2,SAMPLE_RATE)) {
        throw std::runtime_error("setupSoundBuffers(): could not load"
           " SoundBuffer");
       }
       soundBuffers->push_back(buffer);
   }
}
void setupSounds(const std::vector<sf::SoundBuffer>& soundBuffers,
std::vector<sf::Sound>* sounds) {
    sf::Sound sound;
    for (int i = 0; i < static_cast<int>(soundBuffers.size()); i++) {
       sound.setBuffer(soundBuffers[i]);
       sounds->push_back(sound);
    }
}
```

```
// Copyright 2021 Thomas Freeman
#ifndef STRINGSOUND_HPP   //NOLINT
```

```
#define STRINGSOUND_HPP  //NOLINT
#include <math.h>
#include <vector>
#include <string>
#include <random>
#include <SFML/Audio/SoundBuffer.hpp>
#include "CircularBuffer.hpp"
// sample rate is measured in hertz
#define SAMPLE_RATE 44100
#define DECAY_FACTOR 0.996
class StringSound {
public:
   // default constructor
   StringSound();
   // creates stringSound at given frequency
   explicit StringSound(double frequency);
   // Vector gives initial string size and values from Int16
   explicit StringSound(std::vector<sf::Int16> init);
   StringSound(const StringSound& copySS);
   StringSound(StringSound&& moveSS) noexcept;
   ~StringSound();
   // pluck then replaces random values in the buffer
   // tic advances by one step, returning the sample
   // time returns the number of times tic executed
   void pluck();
   void tic();
   sf::Int16 sample();
   int time();
   StringSound& operator=(const StringSound& rightSide);
   StringSound& operator=(StringSound&& rvalue) noexcept;
private:
   CircularBuffer* _rb;
   int _time;
};
#endif  //NOLINT
```

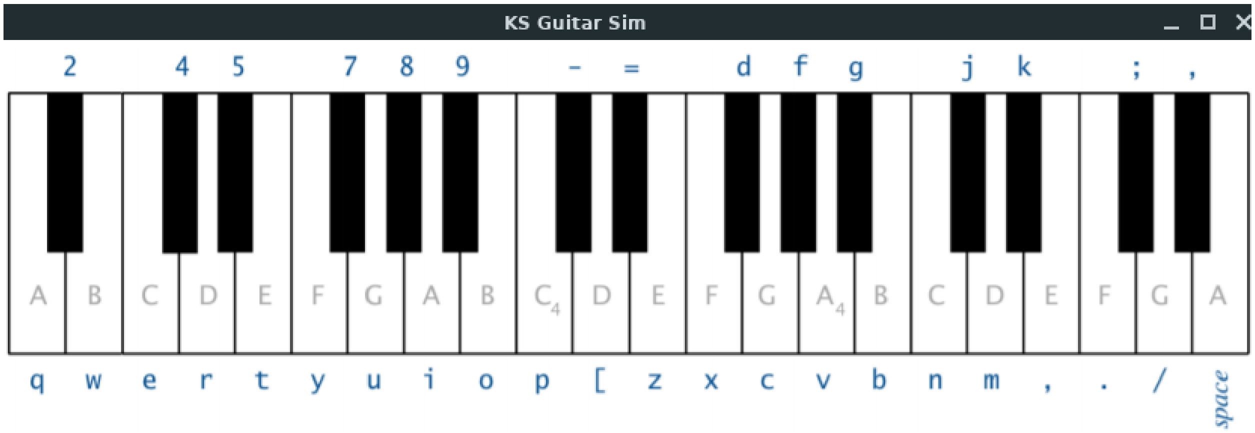
```
// Copyright 2021 Thomas Freeman
#include "StringSound.hpp"
#include <cstdlib>
#include <ctime>
// All listed constructors with try catch blocks
StringSound::StringSound() : _rb(nullptr), _time(0) {}
StringSound::StringSound(double frequency) {
   if (frequency == 0) {
       throw std::invalid_argument("StringSound constructor: frequency"
       " cannot be zero");
   }
   try {
       _rb = new CircularBuffer(ceil(SAMPLE_RATE / frequency));
   }
   catch (std::invalid_argument err) {
       std::cerr << err.what() << std::endl;
   }
   _time = 0;
}
StringSound::StringSound(std::vector<sf::Int16> init) {
   try {
       _rb = new CircularBuffer(init.size());
       for (int i = 0; i < static_cast<int>(init.size()); i++) {
           _rb->enqueue(init[i]);
       }
   }
   catch (std::invalid_argument err) {
       std::cerr << err.what() << std::endl;
   }
   catch (std::runtime_error err) {
       std::cerr << err.what() << std::endl;
   }
   _time = 0;
}
StringSound::StringSound(const StringSound& copySS) {
   _rb = copySS._rb;
   _time = copySS._time;
}
StringSound::StringSound(StringSound&& moveSS) noexcept {
   _rb = moveSS._rb;
   _time = moveSS._time;
   moveSS._rb = nullptr;
   moveSS._time = 0;
}
StringSound::~StringSound() {
  if (_rb != nullptr) { delete _rb; }
   _rb = nullptr;
   _time = 0;
}
// Uses ran for int16_t distribution of sound
void StringSound::pluck() {
   std::random_device device;
   std::mt19937 mt_rand(device());
   std::uniform_int_distribution<int16_t> distribution(INT16_MIN,
   INT16_MAX);
   // buffer is then reset
   _rb->empty();
   _time = 0;
   // and filled with random values
   try {
       for (int i = 0; i < _rb->capacity(); i++) {
           _rb->enqueue(distribution(mt_rand));
       }
   }
   catch (std::runtime_error err) {
       std::cerr << err.what() << std::endl;
   }
}
void StringSound::tic() {
   try {
      int16_t next = DECAY_FACTOR * (_rb->dequeue() + _rb->peek()) / 2;
```

```
        _rb->enqueue(next);
    }
    catch (std::runtime_error err) {
        std::cerr << err.what() << std::endl;
    }
    _time++;
}
sf::Int16 StringSound::sample() {
    sf::Int16 result = 0;
    try {
        result = _rb->peek();
    }
    catch (std::runtime_error err) {
        std::cerr << err.what() << std::endl;
    }
    return result;
}
int StringSound::time() { return _time; }
StringSound& StringSound::operator=(const StringSound& rightSide) {
    if (this != &rightSide) {
        if (_rb != nullptr) { delete _rb; }
        _rb = rightSide._rb;
        _time = rightSide._time;
    }
    return *this;
}
StringSound& StringSound::operator=(StringSound&& rvalue) noexcept {
    if (this != &rvalue) {
        if (_rb != nullptr) { delete _rb; }
        _rb = rvalue._rb;
        _time = rvalue._time;
        rvalue._rb = nullptr;
        rvalue._time = 0;
    }
    return *this;
}
```

```
// copyright 2021 Thomas Freeman
#ifndef CIRCULARBUFFER_HPP   //NOLINT
#define CIRCULARBUFFER_HPP   //NOLINT
#include <stdint.h>
#include <iostream>
#include <exception>
class CircularBuffer {
public:
   explicit CircularBuffer(int capacity);
   // ring buffer is made empty
   // with its own max capacity
   CircularBuffer(const CircularBuffer& copyCB);
   CircularBuffer(CircularBuffer&& moveCB) noexcept;
   ~CircularBuffer();
   int size() const;          // return elements in the buffer
   int capacity() const;      // return capacity of buffer
   bool isEmpty() const;     // checks if size = 0
   bool isFull() const;      // checks if size = capacity
   void enqueue(int16_t x);  // adds an item
   int16_t dequeue();        // deletes an item
   int16_t peek();           // returns the frontmost item
   void empty();             // set size to 0
   CircularBuffer& operator=(const CircularBuffer& rightSide);
   CircularBuffer& operator=(CircularBuffer&& rvalue) noexcept;
private:
   int _size;
   int _capacity;
   int _first;
   int _last;
   int16_t* _buffer;
};
#endif  //NOLINT
```

```
// copyright 2021 Thomas Freeman
#include "CircularBuffer.hpp"
// List of constructors
CircularBuffer::CircularBuffer(int capacity) {
   if (capacity < 1) {
      throw std::invalid_argument("CircularBuffer constructor: capacity"
       " must be greater than zero.");
   }
   _capacity = capacity;
   _size = 0;
   _first = 0;
   _last = 0;
   _buffer = new int16_t[capacity];
}
CircularBuffer::CircularBuffer(const CircularBuffer& copyCB) {
   _capacity = copyCB._capacity;
   if (_capacity > 0) {
       _size = copyCB._size;
       _first = copyCB._first;
       _last = copyCB._last;
       _buffer = new int16_t[_capacity];
       for (int i = 0; i < _size; i++) {
           _buffer[i] = copyCB._buffer[i];
       }
   } else {
       _size = 0;
       _first = 0;
       _last = 0;
       _buffer = nullptr;
   }
}
CircularBuffer::CircularBuffer(CircularBuffer&& moveCB) noexcept {
   _capacity = moveCB._capacity;
   _size = moveCB._size;
   _first = moveCB._first;
   _last = moveCB._last;
   _buffer = moveCB._buffer;
   moveCB._capacity = 0;
   moveCB._size = 0;
   moveCB._first = 0;
   moveCB._last = 0;
   moveCB._buffer = nullptr;
}
CircularBuffer::~CircularBuffer() {
   if (_buffer != nullptr) { delete[] _buffer; }
   _capacity = 0;
   _size = 0;
   _first = 0;
   _last = 0;
   _buffer = nullptr;
}
// main functions in circular buffer
int CircularBuffer::size() const { return _size; }
int CircularBuffer::capacity() const { return _capacity; }
bool CircularBuffer::isEmpty() const { return _size == 0; }
bool CircularBuffer::isFull() const { return _size == _capacity; }
void CircularBuffer::enqueue(int16_t x) {
   if (isFull()) {
       throw std::runtime_error("enqueue: can't enqueue to a full
ring.");
   }
   // we add an item to the buffer then increment
   _buffer[_last] = x;
   _last++;
   if (_last == _capacity) { _last = 0; }
   _size++;
}
int16_t CircularBuffer::dequeue() {
   if (isEmpty()) {
       throw std::runtime_error("dequeue: can't dequeue from an empty"
       " ring.");
   }
   // return the first item then index the new first item
   int16_t result = _buffer[_first];
   _first++;
   if (_first == _capacity) { _first = 0; }
   _size--;
   return result;
}
// peek exception runs, returns first item after
int16_t CircularBuffer::peek() {
   if (isEmpty()) {
      throw std::runtime_error("peek: can't peek from an empty ring.");
   }
   return _buffer[_first];
}
void CircularBuffer::empty() {
   _size = 0;
   _first = 0;
   _last = 0;
```

```
}
// Overloaded assignment operators
CircularBuffer& CircularBuffer::operator=(const CircularBuffer&
rightSide) {
   if (this != &rightSide) {
   if (_buffer != nullptr) { delete[] _buffer; }
   _capacity = rightSide._capacity;
   if (_capacity > 0) {
       _size = rightSide._size;
     _first = rightSide._first;
       _last = rightSide._last;
       _buffer = new int16_t[_capacity];
       for (int i = 0; i < _size; i++) {
           _buffer[i] = rightSide._buffer[i];
           }
       } else {
           _size = 0;
           _first = 0;
           _last = 0;
           _buffer = nullptr;
       }
   }
   return *this;
}
CircularBuffer& CircularBuffer::operator=(CircularBuffer&& rvalue)
noexcept {
   if (this != &rvalue) {
   if (_buffer != nullptr) { delete[] _buffer; }
   _capacity = rvalue._capacity;
   _size = rvalue._size;
   _first = rvalue._first;
   _last = rvalue._last;
   _buffer = rvalue._buffer;
   rvalue._capacity = 0;
   rvalue._size = 0;
   rvalue._first = 0;
   rvalue._last = 0;
   rvalue._buffer = nullptr;
   }
   return *this;
}
```


KS Guitar Sim
2 4 5 7 8 9 - = d f g j k ; ,
A B C D E F G A B C4 D E F G A4 B C D E F G A
q w e r t y u i o p [z x c v b n m , . / space

PS5 – DNA Sequence Alignment

For this assignment Our goal was to choose a way to implement a DNA sequence Analyzer such that we could identify how similar the strings were. We were given multiple ways of doing this and I choose to do the Dynamic Programming approach. This involves the creation of a Matrix that will take the two strings and store sections of each as each section gets analyzed for matches. The Array is filled with the string characters from bottom right to top left and then is traced from top left to bottom right. By Filling and Tracing in this order, the Matrix can store parts of the string that are already analyzed and continue onto the next. Our Main file simply provides the terminal output and syntax for the edit distance and execution time. We also got to see how memory can affect our runtime in this assignment using the valgrind tool to assess our memory usage at runtime. Since My laptop is fairly new it performed above average based upon the availability of memory used for my VM. The example after the code shows the output and execution time of an ecoli strand.

The following Code is for PS5

```
CC = g++
CFLAGS = -std=c++11 -c -g -O2 -Wall -Werror -pedantic
LIBS = -lsfml-system
OBJS = main.o EDistance.o
EXE = EDistance
all : $(EXE)
$(EXE) : $(OBJS)
	$(CC) -o $(EXE) $(OBJS) $(LIBS)
%.o : %.cpp
	$(CC) $(CFLAGS) -o $@ $<
clean :
	\rm $(EXE) $(OBJS)
```

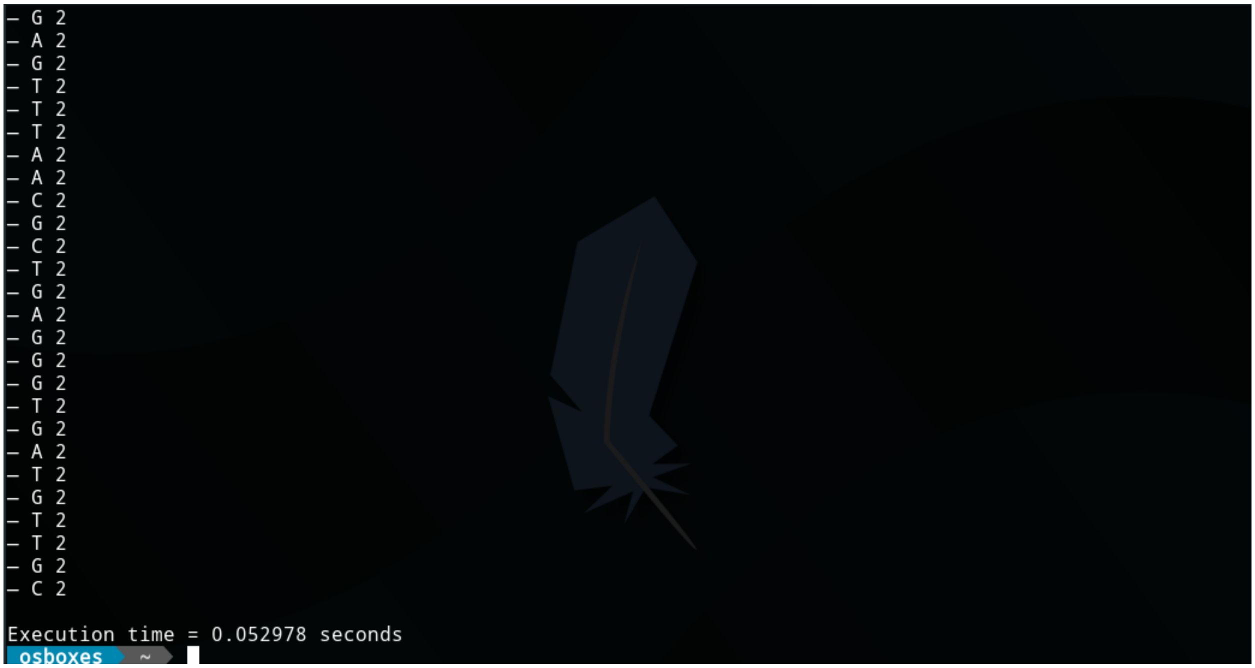
```
// copyright 2021 Thomas Freeman
#include "EDistance.hpp"
#include <SFML/System.hpp>
// command line syntax
```

```
int main(int argc, char* argv[]) {
   if (argc != 1) {
       std::cerr << "usage: ./EDistance < ecoli8000.txt" << std::endl;
       exit(-1);
   }
    EDistance ed;
    std::cin >> ed;
    sf::Clock clock;
    sf::Time t;
    // Print statements for edit distance and execution time
    std::cout << "Edit distance = " << ed.optDistance() << std::endl;
    std::cout << ed.alignment() << std::endl;
    t = clock.getElapsedTime();
    std::cout << "Execution time = " << t.asSeconds() << " seconds" <<
    std::endl;
    return 0;
}

```

```
// copyright 2021 Thomas Freeman
#ifndef EDistance_HPP    //NOLINT
#define EDistance_HPP    //NOLINT
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
class EDistance {
public:
    EDistance();
    // edit distance is made from two strings
    EDistance(std::string x, std::string y);
    // penalties for char's in strings x and y
    static int penalty(char a, char b);
    // returns minimum value of a,b,c
    static int min(int a, int b, int c);
    // fills the matrix
    int optDistance();
   // traces the matrix, returns alignment
    std::string alignment();
    // overload stream operator
    friend std::istream& operator>>(std::istream& in, EDistance& ed);
private:
    // M is for string x, N for string y
    int M;
    int N;
    std::string _x, _y;
    // measures the sequence alignment
    std::vector<std::vector<int> > opt;
};
#endif  //NOLINT
```

```
// Copyright 2021 Thomas Freeman
#include "EDistance.hpp"
```

```
EDistance::EDistance() : M(0), N(0) {}
EDistance::EDistance(std::string x, std::string y) {
    _x = x; _y = y;
    M = static_cast<int>(_x.length());
   N = static_cast<int>(_y.length());
   // initialize vector matrix of ints
   opt = std::vector<std::vector<int> >(M+2, std::vector<int>(N+2, 0));
   // base cases for M and N
   for (int i = M; i >= 0; i--) {
       opt[i][N+1] = opt[i+1][N+1] + 2;
   }
   for (int j = N; j >= 0; j--) {
       opt[M+1][j] = opt[M+1][j+1] + 2;
   }
}
// when both chars are the same return 0, else return 1
int EDistance::penalty(char a, char b) {
   if (a == b)
       return 0;
   else
       return 1;
}
// returns distance based on the minimum int
int EDistance::min(int a, int b, int c) {
   if (b < a && b < c)
       return b;
   if (c < a && c < b)
       return c;
   else
       return a;
}
int EDistance::optDistance() {
   // fill matrix with values
   for (int i = M; i >= 0; i--) {
       for (int j = N; j >= 0; j--) {
           opt[i][j] = min(opt[i+1][j+1] + penalty(_x[i], _y[j]),
           opt[i+1][j] + 2, opt[i][j+1] + 2);
       }
   }
   return opt[0][0];
}
std::string EDistance::alignment() {
   std::string result;
   if (M != 0 && N != 0) {
       // checks for empty string in M and N
       for (int i = 0; i < M; i++) {
           for (int j = 0; j < N; j++) {
               // tracing is done diagonaly
               if (opt[i][j] == opt[i + 1][j + 1] && _x[i] == _y[j]) {
                   result.push_back(_x[i]);
                   result.append(" ");
                   result.push_back(_y[j]);
                   result.append(" 0\n");
                   i++;
               } else if (opt[i][j] == opt[i + 1][j + 1] + 1) {
                   result.push_back(_x[i]);
                   result.append(" ");
                   result.push_back(_y[j]);
                   result.append(" 1\n");
                   i++;
               } else if (opt[i][j] == opt[i + 1][j] + 2) {
                   // move down
                   result.push_back(_x[i]);
                   result.append(" â€” 2\n");
                   i++;
                   j--;
               } else if (opt[i][j] == opt[i][j + 1] + 2) {
                   // move right
                   result.append("â€” ");
                   result.push_back(_y[j]);
                   result.append(" 2\n");
               }
           }
       }
   } else if (M != 0 && N == 0) {
       result.append("String y is empty, returning string x: ");
       result.append(_x);
       result.append("\n");
   } else if (M == 0 && N != 0) {
       result.append("String x is empty, returning string y: ");
       result.append(_y);
       result.append("\n");
   } else {
       result.append("No two strings provided\n");
   }
   return result;
}
std::istream& operator>>(std::istream& in, EDistance& ed) {
   std::string x;
   std::string y;
   in >> x;
   in >> y;
   ed = EDistance(x, y);
   return in;
}
```



```
– G 2
– A 2
– G 2
– T 2
– T 2
– T 2
– A 2
– A 2
– C 2
– G 2
– C 2
– T 2
– G 2
– A 2
– G 2
– G 2
– G 2
– T 2
– G 2
– A 2
– T 2
– G 2
– T 2
– T 2
– G 2
– C 2

Execution time = 0.052978 seconds
osboxes  ~
```

PS6 – Random Writer

This assignment involves the implementation of what's known as the Markov Model. The algorithm can predict the next given number of characters within a given text based upon our K grams or sets of characters contained within the code. Our text file is taken as a parameter and then we map all our K grams from said text file, in my case I used the Amendments text file. A map can be seen as a table that contains the characters represented by each K gram. This map is created using the Mersenne twister number generator along with the compiler based Krand function to help select proceeding characters. Our new string is then created based upon the generated probabilities for proceeding characters. Our main method Text Writer then analyzes our string and creates a string L that recreates the text up to a certain point. For the Alphabet text file using L = 26 results in printing out the entire alphabet. My example below shows an output of the amendments.txt file.

Here's the following code for PS6

```
CC = g++
CFLAGS = -std=c++11 -c -g -O1
LIBS = -lboost_unit_test_framework
OBJS = TextWriter.o RandWriter.o
EXE = TextWriter
all : $(EXE)
$(EXE) : $(OBJS)
	$(CC) -o $(EXE) $(OBJS) $(LIBS)
%.o : %.cpp
	$(CC) $(CFLAGS) -o $@ $<
```

```
clean :
	\rm $(EXE) $(OBJS)
```

```
// copyright 2021 Thomas Freeman
#include "RandWriter.h"
RandWriter::RandWriter(std::string text, int k) {
    _text = text;
    _k = k;
    if (_text.length() < static_cast<unsigned int>(_k)) {
        throw std::invalid_argument("RandWriter(string text, int k):
order k"
        " must be less than or equal to text length.");
    }
    // Markov table
    unsigned int pos = 0;
    for (unsigned int i = 0; i < _text.length(); i++) {
        std::string k_gram;
        std::map<char, int> f_table;
        // maps corresponding char to its frequency
        // parses the kgrams text and gets characters
        for (unsigned int j = i; j < i + _k; j++) {
            if (j >= _text.length()) {
                pos = j - _text.length();
            } else {
                pos = j;
            }
            k_gram += _text.at(pos);
        }
        // sets up next freq table
        pos++;
        if (pos >= _text.length()) { pos -= _text.length(); }
        f_table.insert(std::make_pair(_text.at(pos), 0));
        // k_gram and frequency tables are put into the map
        if (mtable.count(k_gram) == 0) {
            mtable.insert(std::make_pair(k_gram, f_table));
        }
        // update next char in freq table
        mtable[k_gram][_text.at(pos)]++;
    }
}
int RandWriter::kOrder() const { return _k; }
std::string RandWriter::getText() const { return _text; }
std::map<std::string, std::map<char, int>> RandWriter::getMTable()
const {
    return mtable;
}
int RandWriter::freq(std::string k_gram) const {
    if (k_gram.length() < static_cast<unsigned int>(_k)) {
       throw std::runtime_error("freq(string k_gram): k_gram must be of"
        " length greater than or equal to order k.");
    }
    int count = 0;
    for (unsigned int i = 0; i < _text.length(); i++) {
        unsigned int pos = 0;
        std::string kg;
        // parse input text for kgrams
        for (unsigned int j = i; j < i + _k; j++) {
            // get characters for kgrams
            if (j >= _text.length()) {
                pos = j - _text.length();
            } else {
                pos = j;
            }
            kg += _text.at(pos);
        }
        if (k_gram == kg) { count++; }
    }
    return count;
}
int RandWriter::freq(std::string k_gram, char c) const {
    if (k_gram.length() < static_cast<unsigned int>(_k)) {
  throw std::runtime_error("freq(string k_gram, char c): k_gram must be"
        " of length greater than or equal to order k.");
    }
    return mtable.at(k_gram).at(c);
}
// exceptions for k_gram
char RandWriter::kRand(std::string k_gram) const {
    if (k_gram.length() < static_cast<unsigned int>(_k)) {
      throw std::runtime_error("kRand(string k_gram): k_gram must be of"
        " length greater than or equal to order k.");
    }
    if (mtable.count(k_gram) == 0) {
        throw std::runtime_error("kRand(string k_gram): k_gram does not"
        " exist.");
    }
    // next chars stored as a string
    std::string alphabet;
    for (auto const &var1 : mtable) {
        if (var1.first == k_gram) {
            for (auto const &var2 : var1.second) {
                alphabet += var2.first;
            }
        }
    }
    std::random_device device;
    std::mt19937 mt_rand(device());
    std::uniform_int_distribution<int> distribution(0, alphabet.length()
    - 1);
    return alphabet[distribution(mt_rand)];
}
std::string RandWriter::generate(std::string k_gram, int L) const {
    if (k_gram.length() < static_cast<unsigned int>(_k)) {
  throw std::runtime_error("generate(string k_gram, int L): k_gram must"
         " be of length greater than or equal to order k.");
    }
    std::string generated = k_gram;
    // new characters generated based on k_gram
    for (int i = _k; i < L; i++) {
        generated += kRand(generated.substr(i - _k, _k));
    }
    return generated;
}
std::ostream& operator<<(std::ostream& out, const RandWriter&
randwriter) {
    out << "Markov Model\tOrder: " << randwriter._k << std::endl;
    out << "k_gram:\tfrequency:\tfrqncy of next char:\tprob of next
char:" << std::endl;

    for (auto const &var1 : randwriter.mtable) {
        // var1.first = k_gram
        out << var1.first << "\t";
        out << randwriter.freq(var1.first) << "\t\t";
        for (auto const &var2 : var1.second) {
        // var2.first = next char
        // var2.second = data
            out << var2.first << ":" << var2.second << " ";
        }
        out << "\t\t\t";
        for (auto const &var2 : var1.second) {
            out << var2.first << ":" << var2.second << "/" <<
            randwriter.freq(var1.first) << " ";
        }
        out << std::endl;
    }
    return out;
}
```

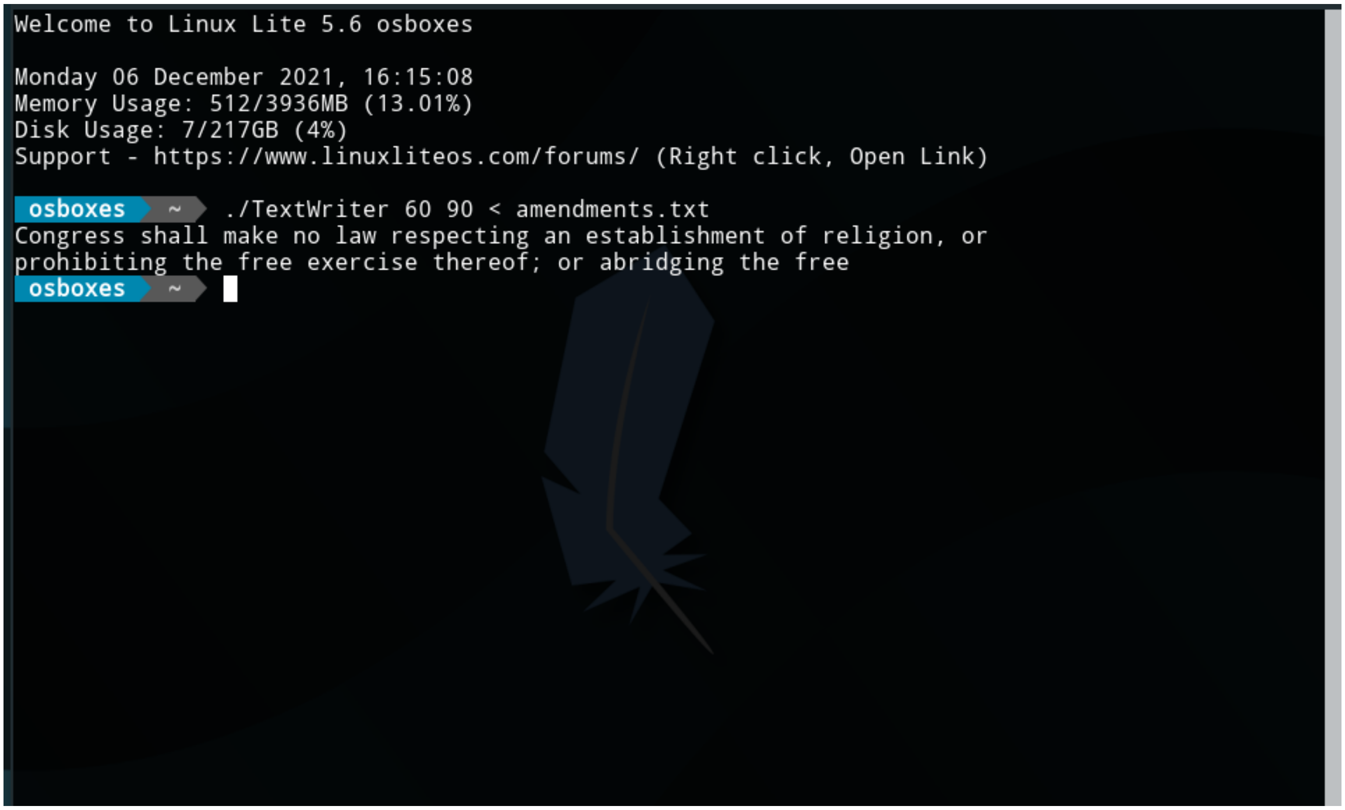
```
// copyright 2021 Thomas Freeman
#ifndef RANDWRITER_H  //NOLINT
#define RANDWRITER_H  //NOLINT
#include <iostream>
#include <string>
#include <map>
#include <exception>
#include <utility>
#include <random>
class RandWriter {
public:
    // creates markov model of order k based on the text file
    RandWriter(std::string text, int k);
    // returns Korder
    int kOrder() const;
    // returns input text
    std::string getText() const;
    // Returns the table map
    std::map<std::string, std::map<char, int>> getMTable() const;
    // returns the number of times K_gram occours in the text
    // throws exception if K_gram is not length K
    int freq(std::string k_gram) const;
    // returns how many times char c follows k_gram
```

```
   // throws exception if k_gram is not length K
   int freq(std::string k_gram, char c) const;
   // returns random character following k_gram
   // throws exception if k_gram is not length K
   // or if there's no k_gram
   char kRand(std::string k_gram) const;
   // generates string of length L by simulating the markov chain
   // first k characters of the new string are K_gram since L is close
to K
   // throws exception otherwise
   std::string generate(std::string k_gram, int L) const;
   // overloaded stream insertion operator to show the models state
   friend std::ostream& operator<<(std::ostream& out, const RandWriter&
   randwriter);
private:
   int _k;             // order of Markov Model
   std::string _text;  // text to analyze
   // map of k_gram to map of frequency of next char
   std::map<std::string, std::map<char, int>> mtable;
};
#endif   //NOLINT
```

```
// copyright 2021 Thomas Freeman
#include "RandWriter.h"
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Main
#include <boost/test/unit_test.hpp>
BOOST_AUTO_TEST_CASE(base_test) {
   std::cout << "Test Case 1 " <<
   std::endl;
   int k = 2;
   std::string str = "gagggagaggcgagaaa";
   RandWriter randwriter(str, k);
   std::cout << "Printing out Markov Table for string:\n" <<
   str << std::endl << std::endl;
   std::cout << randwriter << std::endl;
       std::cout << "Testing kOrder and freq functions" << std::endl;
       BOOST_REQUIRE(randwriter.kOrder() == k);
       BOOST_REQUIRE(randwriter.freq("gg") == 3);
       BOOST_REQUIRE(randwriter.freq("ga", 'g') == 4);
       std::cout << "Testing kRand function" << std::endl;
       char rand = randwriter.kRand("aa");
       BOOST_REQUIRE(rand == 'a' || rand == 'g');
       std::cout << "Testing generate function" << std::endl <<
std::endl;
       BOOST_REQUIRE(randwriter.generate("ga", 10).length() == 10);
}
BOOST_AUTO_TEST_CASE(exception_test) {
   std::cout << " Test Case 2 " <<
   std::endl;
   std::cout << "Testing construction exception: RandWriter('ADF', 4)"
<< std::endl;
   BOOST_REQUIRE_THROW(RandWriter("ADF", 4), std::invalid_argument);
```

```
    std::cout << "Testing function exceptions" << std::endl;
    RandWriter testMtable("abc", 3);
    BOOST_REQUIRE_THROW(testMtable.freq("a"), std::runtime_error);
    BOOST_REQUIRE_THROW(testMtable.freq("ab", 'b'), std::runtime_error);
    BOOST_REQUIRE_THROW(testMtable.kRand("g"), std::runtime_error);
}
```

```
// copyright 2021 Thomas Freeman
#include "RandWriter.h"
#include <fstream>
int main(int argc, char *argv[]) {
   if (argc != 3) {
       std::cerr << "Usage: ./TextWriter k L < input.txt" << std::endl;
       exit(-1);
   }
   int k = std::atoi(argv[1]);
    int L = std::atoi(argv[2]);
    int count = 0;
    int length = 0;
    std::string input;
    std::string output;
    // reads line by line of the input and generates
    // characters pseudorandomly
    while (std::getline(std::cin, input) && count < L) {
        if (input.length() > static_cast<unsigned int>(k)) {
            try {
                RandWriter randwriter(input, k);
                if (static_cast<int>(input.length()) > L) {
                    length = L;
                } else if (static_cast<int>(input.length()) + count > L)
               {
                    length = L - count;
                } else {
                    length = input.length();
                }
                output = randwriter.generate(input.substr(0, k),
length);
                count += output.length();
                std::cout << output << std::endl;
            }
            catch (std::invalid_argument err) {
                std::cerr << err.what() << std::endl;
                exit(-1);
            }
            catch (std::runtime_error err) {
                std::cerr << err.what() << std::endl;
                exit(-1);
            }
        }
    }
    return 0;
}
```



```
Welcome to Linux Lite 5.6 osboxes

Monday 06 December 2021, 16:15:08
Memory Usage: 512/3936MB (13.01%)
Disk Usage: 7/217GB (4%)
Support - https://www.linuxliteos.com/forums/ (Right click, Open Link)

osboxes ~ ./TextWriter 60 90 < amendments.txt
Congress shall make no law respecting an establishment of religion, or
prohibiting the free exercise thereof; or abridging the free
osboxes ~
```

PS7 – Kronos Time Clock

For Our last assignment we would be using the Kronos Time Clock in order to see how the implementation of regular expressions could help in parsing the Log Files of the Clock. The Log files are mainly used to determine points of failure, determined at the date and time. We would need to implement regular expressions to scan the file to report each time the clock is booted and finished with its boot. Printing this in a separate report file (.rpt). When the Boot Start message in the Log file is scanned the Date and Time is recorded using our regular expression. Upon completion of the boot, the Date and Time is recorded again, being put into our Report files. The Example after the code shows the first report file after scanning the first log file.

The Following Code is for PS7

```
CC = g++
CFLAGS = -std=c++11 -c -g -O1 -Wall -Werror -pedantic
LIBS = -lboost_regex -lboost_date_time
OBJS = main.o
EXE = ps7
all : $(EXE)
$(EXE) : $(OBJS)
```

```
    $(CC) -o $(EXE) $(OBJS) $(LIBS)
%.o : %.cpp
    $(CC) $(CFLAGS) -o $@ $<
clean :
    \rm $(EXE) $(OBJS)
```

```
// Copyright 2021 Thomas Freeman
#include <iostream>
#include <fstream>
#include <string>
#include <exception>
#include <boost/regex.hpp>
#include <boost/date_time.hpp>
// boost libraries for regular expression
using boost::regex;
using boost::regex_search;
using boost::smatch;
using boost::posix_time::ptime;
using boost::posix_time::time_duration;
using boost::posix_time::time_from_string;
int main(int argc, char* argv[]) {
    int lineNum = 1, bootStartCount = 0, bootCompleteCount = 0;
    bool bootStarted = false;
    const std::string bootStartMsg = "(log.c.166) server started";
    const std::string bootCompleteMsg = "oejs.AbstractConnector:Started"
    "SelectChannelConnector@0.0.0.0:9080";
        // this regular expressions captures the date and time
        std::string s;
        std::string fileName;
        std::ifstream inputFile;
        std::ofstream outputFile;
        regex e("^\\d{4}[-](0[1-9]|1[012])[-](0[1-9]|[12][0-9]"
        "|3[01])\\s\\d{2}[:]\\d{2}[:]\\d{2}");
        smatch m;
        // t1 and t2 determine start and complete times
        ptime t1, t2;
    if (argc != 2) {
        std::cerr << "Usage: ./ps7 device1_intouch.log" << std::endl;
        return -1;
    }
    // setup file I/O
    inputFile.open(argv[1]);
    if (!inputFile.is_open()) {
        std::cerr << "Could not open file: " << argv[1] << std::endl;
        return -1;
    }
    s = fileName = argv[1];
    outputFile.open(s.append(".rpt.tmp"));
    fileName = fileName.substr(fileName.find_last_of("\\/") + 1);
    // scans boot info from log file, transfers info to report file.
    while (std::getline(inputFile, s)) {
        if (bootStarted) {
            if (s.find(bootCompleteMsg) != std::string::npos) {
                // device boot completes
                regex_search(s, m, e);
                // we then get the date and time
                t2 = ptime(time_from_string(m[0]));
                // and total boot time
                time_duration td = t2 - t1;
                outputFile << lineNum << "(" << fileName << ") " << m[0]
                << " Boot Completed" << std::endl
                << "\tBoot Time: " << td.total_milliseconds() << "ms"
                << std::endl << std::endl;
                bootStarted = false;
                bootCompleteCount++;
            } else if (s.find(bootStartMsg) != std::string::npos) {
                // boot start reports after unsuccessful boot
                regex_search(s, m, e);
                t1 = ptime(time_from_string(m[0]));
                outputFile << " Incomplete boot" << std::endl <<
                std::endl
                << " Device boot" << std::endl
                << lineNum << "(" << fileName << ") " << m[0]
                << " Boot Start" << std::endl;
                bootStartCount++;
            }
        } else if (s.find(bootStartMsg) != std::string::npos) {
            // boot start reports again
            regex_search(s, m, e);
            t1 = ptime(time_from_string(m[0]));
            outputFile << "=== Device boot ===" << std::endl
            << lineNum << "(" << fileName << ") " << m[0]
            << " Boot Start" << std::endl;
            bootStarted = true;
            bootStartCount++;
        }
        lineNum++;
    }
    inputFile.close();
    outputFile.close();
    // add .rpt and .tmp to report file
    s = argv[1];
    s.append(".rpt");
    outputFile.open(s);
    s.append(".tmp");
    inputFile.open(s);
    if (!inputFile.is_open()) {
        std::cerr << "Could not open file: " << s << std::endl;
        return -1;
    }
    outputFile << "Device Boot Report" << std::endl << std::endl
    << "InTouch log file: " << fileName << std::endl
    << "Lines Scanned: " << lineNum - 1 << std::endl << std::endl
    << "Device boot count: initiated: " << bootStartCount << ",
completed: "
    << bootCompleteCount << std::endl << std::endl << std::endl;
    outputFile << inputFile.rdbuf();    // copies the data
    inputFile.close();
    outputFile.close();
    // removes the temp file
    if (std::remove(s.c_str()) != 0) {
        std::cerr << "Error deleting temp file: " << s << std::endl;
        return -1;
    }
    return 0;
}
```

```
Device Boot Report
InTouch log file: device1_intouch.log
Lines Scanned: 443838
Device boot count: initiated: 6, completed: 6
=== Device boot ===
435369(device1_intouch.log) 2014-03-25 19:11:59 Boot Start
435759(device1_intouch.log) 2014-03-25 19:15:02 Boot Completed
      Boot Time: 183000ms
=== Device boot ===
436500(device1_intouch.log) 2014-03-25 19:29:59 Boot Start
436859(device1_intouch.log) 2014-03-25 19:32:44 Boot Completed
      Boot Time: 165000ms
=== Device boot ===
440719(device1_intouch.log) 2014-03-25 22:01:46 Boot Start
440791(device1_intouch.log) 2014-03-25 22:04:27 Boot Completed
      Boot Time: 161000ms
=== Device boot ===
440866(device1_intouch.log) 2014-03-26 12:47:42 Boot Start
441216(device1_intouch.log) 2014-03-26 12:50:29 Boot Completed
      Boot Time: 167000ms
=== Device boot ===
442094(device1_intouch.log) 2014-03-26 20:41:34 Boot Start
442432(device1_intouch.log) 2014-03-26 20:44:13 Boot Completed
      Boot Time: 159000ms
=== Device boot ===
443073(device1_intouch.log) 2014-03-27 14:09:01 Boot Start
443411(device1_intouch.log) 2014-03-27 14:11:42 Boot Completed
      Boot Time: 161000ms
```