Ryan Casey

COMP IV Sec 203: Project Portfolio

Spring 2022

Contents:

Time to complete all assignments: 78 Hours.

1 PS0: Hello World with SFML

1.1 Objective

The main objective of this assignment was to do a first time setup of our development environment that we would be using for the duration of the semester. For my environment, this included setting up a virtual machine that would run Linux utilizing Unbuntu 64bit, the acquisition of the Safe Fast Media Library (SFML), and an implementation of a ”Hello, World” program. Once all steps were completed, the demo ”Hello, World” program was extended to display a picture to the screen on an on screen window via SFML.

1.2 Outcome

I was able to successfully accomplish all steps of the initial Linux environment setup by way of using Oracle VM Virtual Box Manager. In addition, I was able to implement a program that would display a desired sprite to the screen that would also respond and move to user keyboard inputs.

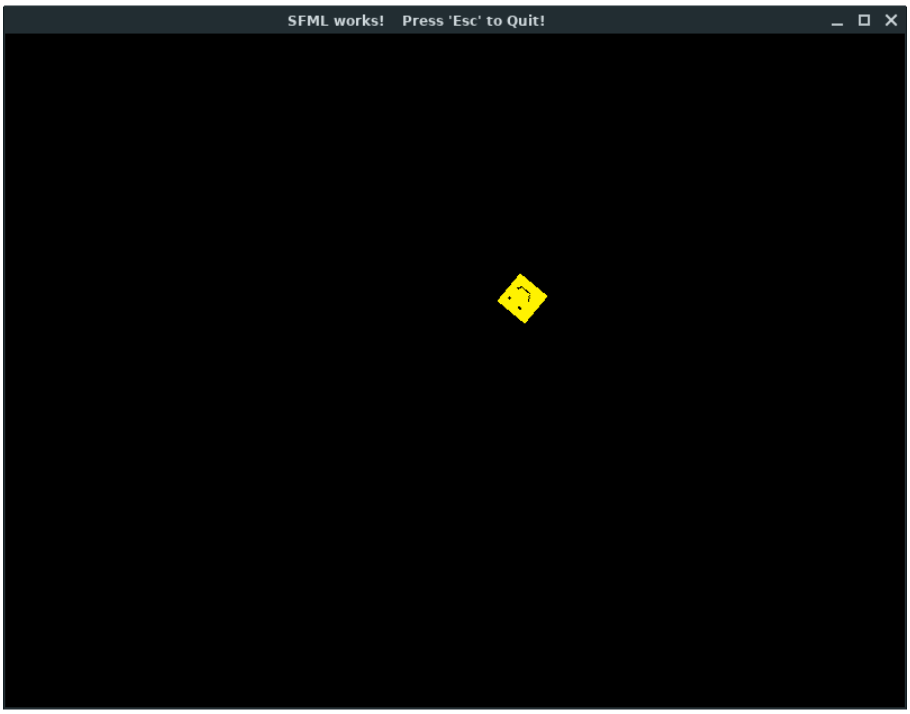

Figure 1.1: Screenshot of PS0 program output.

1.3 Implementation

My program was implemented using multiple objects included in the SFML/Graphics library. These objects included sf::RenderWindow, sf::Texture, sf::Sprite, and sf::Event.

The sf::RenderWindow object was used to create the actual on screen window that can be seen in Figure 1.1. This object took multiple parameters that allowed for the adjustment of the window size and window name.

The sprite that is visible on screen in Figure 1.1 was created by loading an image of choice into an sf::Texture object and then storing said texture to an sf::Sprite object.

Lastly, an event loop was created using the sf::Event object and having the previously mentioned sf::RenderWindow object continually poll this event object every frame to detect specific events from the user or program. In addition to polling for events, the event loop also would instruct the render window to clear(), draw(), and display() at the end of every frame as well. This would in effect update the appearance of our window and sprite every frame.

Using this event loop I was able to allow the user to control the position and rotation of the on screen sprite. This was accomplished by checking every frame for another event object, sf::Keyboard::isKeyPressed. Using this object and keyboard key code, I was able to allow for the transformation of the sprite depending on what keystrokes were seen from the user.

1.4 Lessons Learned

The main things that I took away from this assignment were the steps needed to setup a virtual machine and familiarizing myself with Ubuntu as this was my first experience using either. After getting things setup, I was pleasantly surprised how easy it was to install the required programs and libraries in Ubuntu all via the command terminal. Everything felt very plug and play. This definitely changed my preconceived notions regarding using Ubuntu and has surely cemented itself as an environment I would gladly use in the future.

As stated before, this was my first time using a virtual machine and so, some of the more challenging parts of this assignment for me personally was figuring out how enable CPU virtualization on my personal computer. In order to allow this I had to disable a few safeguards in my Windows OS and also enable virtualization directly in my PC BIOS settings. After some tinkering though, I was able to get everything working as intended.

1.5 Code Listing

1.5.1 Driver Code

main.cpp

```cpp
/**
* PS0 - "SFML Hello World"
* Instructor: James Daly
* Todays Date: 1/20/2022
* Due Date : 1/24/2022
* Program Description:
* Small game window created to test SFML Library.
* Includes minor funcationality including movement of sprive via arrow keys.
* Rotation of the sprite via Q/E keys.
* Closure of game window using 'Esc'.
*
* Created by: Ryan Casey
*/

#include <SFML/Graphics.hpp>


int main()
{
    //Game window object
    sf::RenderWindow window(sf::VideoMode(800, 600), "SFML works!    Press 'Esc' to Quit!");
    // Cap framerate to 60fps
    window.setFramerateLimit(60);


    // Load a texture and display to a sprite.
    sf:: Texture texture;
    if(!texture.loadFromFile("sprite.png"))
        return EXIT_FAILURE;
    sf::Sprite sprite(texture);
    sprite.setPosition(400,300);
    sprite.setOrigin(16,16);

    // Start Game Loop
    while (window.isOpen())
    {
        // Process events
        sf::Event event;
        while (window.pollEvent(event))
        {
            // Close window: exit
            if (event.type == sf::Event::Closed)
                window.close();
        }

        //Check for keyboard inputs
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
        {
                // Move sprite to the left by 2 pixels per frame.
```

```
            sprite.move(-2.0,0.0);
        }
         if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
        {
                // Move sprite to the right by 2 pixels per frame.
            sprite.move(2.0,0.0);
        }
         if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up))
        {
                // Move sprite up by 2 pixels per frame.
            sprite.move(0.0,-2.0);
        }
         if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down))
        {
                // Move sprite down by 2 pixels per frame.
            sprite.move(0.0,2.0);
        }
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Q))
        {
                // Rotate sprite counter clockwise 5 degree per frame.
            sprite.setRotation(sprite.getRotation() - 5);
        }
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::E))
        {
                // Rotate sprite counter clockwise 5 degree per frame.
            sprite.setRotation(sprite.getRotation() + 5);
        }

         if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape))
        {
                // quit when 'Esc' is pressed...
            window.close();
        }



        // Clear the screen
        window.clear();

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

        // Update window
        window.display();
    }

    return EXIT_SUCCESS;
}
```

2 PS1: PhotoMagic

2.1 Objective

The objective of this assignment was to implement a program that would encrypt a provided image file and output said image in its encrypted form. Likewise, the encrypted output image would also need to be able to be decrypted by the same program and be returned to its original state.

2.2 Outcome

My implementation of this program was successful in encrypting and decrypting the supplied input image. The program itself is provided with an input image, a desired output file name, and a bitstring that acts as an encrpytion key.

The encryption output file was stored using the .png file format as it is a lossless compression format. This was done to allow the output file to be decrypted using the same code.

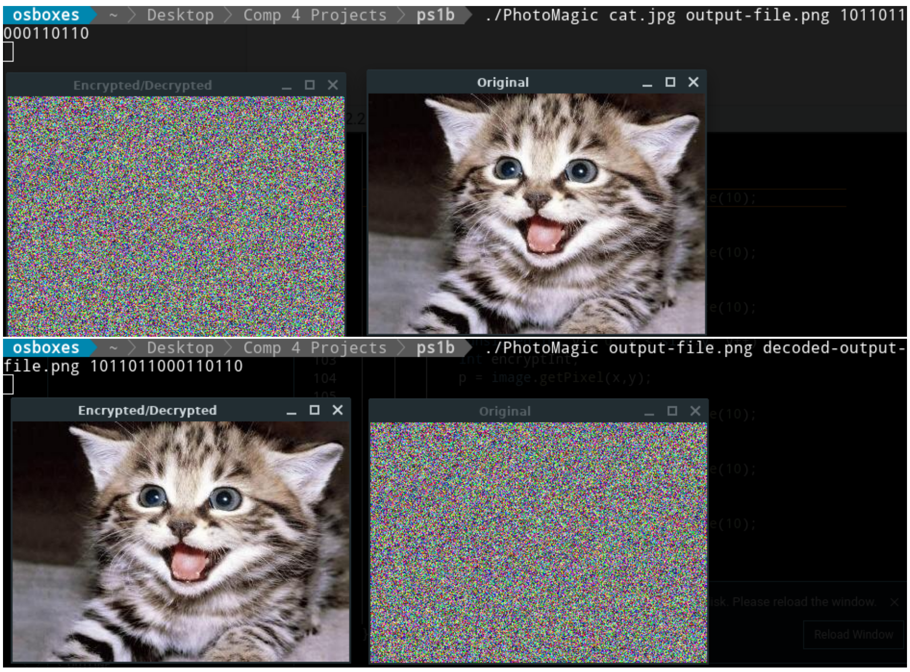

Figure 2.1: Results of encryption/decryption. Input file is shown on right. Output file is shown on left.

2.3 Implementation

This program was implemented by encryption of the image using a One-time pad. I was able to accomplish this by the creation of a class called FibLFSR which emulates a Fibonacci Linear Forward Shift Register.

This FibLFSR operates by a populating a vector with a pre-chosen bit-string that acts as our encryption key. This register will then preform an XOR operation on a set of 3 specified "tap" bits and store the results of that operation for use as a new bit in our register. After the 3 tap bits have been computed together, the register is "left-shifted" by 1 bit and the results of our new bit are then stored in the new spot that was just made by left-shifting the register. This also in effect removed the previous most significant bit of the register.

The FibLFSR will continually generate a new bit via XOR'ing the tap bits, and left-shifting by 1 bit to create room for the new generated bit every time the step() function of the FibLFSR class is called. Using this functionality, the generate(int n) function of the FibLSFR will generate an integer value by calling step() n times and then collecting the least significant bit of the register every time step() is called. After this, the results of collecting the LSB at each step are then used to generate and return new integer value.

This newly generated value from the FibLSFR.generate() function is then used to encrypt the image by performing an XOR operation of the color values of each pixel of the original image with the results returned by generate(). This method of using a predefined encryption key allowed for the image to be decrypted in exactly the same fashion.

2.4 Lessons Learned

The main concepts that I learned in this assignment were the concept of a One-time Pad and cryptography as a whole.

A One-time Pad is a method of encryption where a predefined encryption key is created for a message to be encoded. One method of OTP encryption would be to preform an XOR operation on every bit of the original message with every bit of the key to generate a fully encoded message. This message can then only be decoded by preforming the same XOR operation of the encoded message against the key.

In real world use, if a OTP key is as long as the message to be encoded, is sufficiently random, and is only used securely by both parties, then the encoded message is effectively uncrackable. The main draw backs of OTP encryption however are that a physical key needs to exist somewhere until it is used by both parties. This means that the key itself can become compromised in transit or after use if the key is not destroyed by both parties immediately after use.

2.5 Code Listing

2.5.1 Makefile

```
CC = g++
CFLAGS = -Wall -Werror -pedantic --std=c++14
LIBS = -lsfml-graphics -lsfml-window -lsfml-system
DEPS = FibLFSR.h

all: PhotoMagic FibLFSR.o PhotoMagic.o

PhotoMagic: PhotoMagic.o FibLFSR.o
	$(CC) $(CFLAGS) -o $@ $^ $(LIBS)

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

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

clean:
	rm *.o PhotoMagic
```

2.5.2 Driver Code

PhotoMagic.cpp

```
/***********************************************************
 *Name: Ryan Casey
 *Course name: <COMP.2040>
 *Assignment: PS1B - PhotoMagic
 *Instructor's name: <Dr. James Daly>
 *Date: <2/3/2022>
 *Time to complete assignment: 5 hours.
 *Sources Of Help:
 https://www.sfml-dev.org/documentation/2.5.1/classsf_1_1Color.php
 https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/
 Included pixels.cpp file.
***********************************************************/
#include <iostream>
#include <SFML/System.hpp>
#include <SFML/Window.hpp>
#include <SFML/Graphics.hpp>
#include "FibLFSR.h"



void transform(sf::Image& image, FibLFSR* gen);



int main(int argc, char *argv[])
{
    FibLFSR generator(argv[3]);
```

```
    // load image from file to be encrypted with transform()
    sf::Image encryptMe;
    encryptMe.loadFromFile(argv[1]);

    // original image for comparison
     sf::Image origImage;
    origImage.loadFromFile(argv[1]);

     // encrypt the image and output it to working directory using the filename from argv[2].
     transform(encryptMe, &generator);
    encryptMe.saveToFile(argv[2]);

    //size will hold the (x,y) pair representing the dimensions of the image
    sf::Vector2u size = encryptMe.getSize();

    // create windows to display both the original and encrypted/decrypted image.
    sf::RenderWindow window1(sf::VideoMode(size.x, size.y), "Original");
    sf::RenderWindow window2(sf::VideoMode(size.x, size.y), "Encrypted/Decrypted");

    // load images into textures. Then textures into sprites.
    sf::Texture texture1;
    texture1.loadFromImage(origImage);

    sf::Texture texture2;
    texture2.loadFromImage(encryptMe);

    sf::Sprite sprite1;
    sprite1.setTexture(texture1);

    sf::Sprite sprite2;
    sprite2.setTexture(texture2);


    // render windows and images.
    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(sf::Color::White);
       window1.draw(sprite1);
```

```
      window1.display();

      window2.clear(sf::Color::White);
      window2.draw(sprite2);
      window2.display();
   }

    return 0;
}

// transforms image using FibLFSR as means to encrypt/decrypt the image.
void transform(sf::Image& image, FibLFSR* generator)
{
   // declare a pixel for modification
   sf::Color p;

   // size will hold the (x,y) pair representing the dimensions of the ↩
      image
   sf::Vector2u size = image.getSize();

   // encrypt the image pixel by pixel.
   for (unsigned int x = 0; x < size.x; x++) {
      for (unsigned int y = 0; y < size.y; y++) {
         int encryptInt;
            p = image.getPixel(x,y);

         encryptInt = generator->generate(10);
         p.r = p.r ^ encryptInt;

         encryptInt = generator->generate(10);
         p.g = p.g ^ encryptInt;

         encryptInt = generator->generate(10);
         p.b = p.b ^ encryptInt;

         image.setPixel(x, y, p);
      }
   }
}
```

2.5.3 Interface Files (.hpp)

FibLFSR.hpp

```cpp
#ifndef FIBLFSR_H
#define FIBLFSR_H
#include <iostream>
#include <string>
#include <vector>

class FibLFSR {
public:
    FibLFSR(std::string seed); //constructor - Will populate LFSR based on seed.
    ~FibLFSR();                 //destructor
    int step();                //simulate 1 step in LFSR. Return LSB.
    int generate(int k);       //simulate k steps and return k-bit integer.

    friend std::ostream& operator<< (std::ostream&, const FibLFSR fibLFSR);

private:
    std::string _seed;
    std::vector<int> _bitArray;
    int _bitInteger;
};
#endif
```

2.5.4 Implementation Files (.cpp)

FibLFSR.cpp

```cpp
#include "FibLFSR.h"

//constructor
FibLFSR:: FibLFSR(std::string seed)
{
    //parse bit string char by char and push onto vector
          int bitsRead = 0;

        for(char const ch: seed)
      {
          // Convert bit from char to int.
          int i = ch - '0';

          // Verify bit is valid.
              if (i != 0 && i != 1)
              {
                  throw std::invalid_argument("Invalid bits read in seed. Bits must be 1 or 0.");
              }

          // Detect total bits read from seed. Stop reading at 16 bits.
          bitsRead++;
```

```
          if (bitsRead > 16) {
              break;
          }

          // Push valid bit onto array
          _bitArray.push_back(i);
      }

      // Pad out remainder of empty bits with 0 if seed length is < 16
      if (_bitArray.capacity() < 16)
      {

          for(int i = _bitArray.capacity(); i < 16; i++)
          {
              auto it = _bitArray.begin();
              _bitArray.emplace (it, 0);
          }
      }
}

//destructor
FibLFSR:: ~FibLFSR(){}

//simulate one step of FibLFSR and return generated bit.
int FibLFSR:: step()
{
  //generate new bit to be added by:
  // bit 15 xor bit 13 = result
  // result xor bit 12 = result
  // result xor bit 10 = newBit

  int newBit;
  newBit = ( ( (_bitArray.at(15 - 15)
             ^ _bitArray.at(15 - 13) )
             ^ _bitArray.at(15 - 12) )
             ^ _bitArray.at(15 - 10) );


  _bitArray.push_back(newBit);
  _bitArray.erase(_bitArray.begin());

  return _bitArray.back();
}

//return an integer value based on the bit string generated by repeating
//step(k) times.
int FibLFSR:: generate(int k)
{

  int _bitInteger = 0;
  if (k > 31)
  {
    k = 31; // cap generate at 31 to prevent overflow.
  }

  for (int i = 0; i < k; ++i)
  {
    _bitInteger *= 2;
```

```
    if(step())
    {
      _bitInteger += 1;
    }
  }
  return _bitInteger;

}

//extraction overload
std::ostream& operator<< (std::ostream& os, const FibLFSR fibLFSR)
{
  std::string stringArray = "";
  for(int i: fibLFSR._bitArray){
    char ch = i + '0';
    stringArray += ch;

  }
  os << stringArray;
  return os;
}
```

2.5.5 Test Files (.cpp)

test.cpp

```
// Dr. Rykalova
// test.cpp for PS1a
// updated 1/31/2020

#include <iostream>
#include <string>
#include<sstream>

#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);
}

```

```
BOOST_AUTO_TEST_CASE(constructorTests) {

  // Testing of register padding functionality. Any seed under 16 bits is padded with 0's to fill the register to 16 bits.
  FibLFSR ConstructorTest1("1");
  std::stringstream testStream1;
  testStream1 << ConstructorTest1;
  std::string testString1;
  testStream1 >> testString1;
  BOOST_REQUIRE(testString1.length() == 16);

  // Testing seed truncation functionality. Any seed longer than 16 bits is not read past bit 16.
  FibLFSR ConstructorTest2("0000111110000011111000001111");
  std::stringstream testStream2;
  testStream2 << ConstructorTest2;
  std::string testString2;
  testStream2 >> testString2;
  BOOST_REQUIRE(testString2.length() == 16);

  // Testing invalid seed functionality. Constructor will throw invalid_argument exception if seed contains non 1 or 0 bits.
  BOOST_REQUIRE_THROW(FibLFSR ConstructorTest3("0000000000000002"); , std::invalid_argument);
}

  BOOST_AUTO_TEST_CASE(methodTests) {
  // Testing overflow protection for generate(k). k > 31 will cause overflow and become negative.
  // Any value k > 31 is capped at 31.

  FibLFSR Overflow("1010101010101010");
  BOOST_REQUIRE(Overflow.generate(31) >= 0);
  BOOST_REQUIRE(Overflow.generate(32) >= 0);

  // Testing underflow protection. Any generate(k) with k < 0 will always generate a 0 result.
  BOOST_REQUIRE(Overflow.generate(-1) == 0);
  BOOST_REQUIRE(Overflow.generate(-10) == 0);

  // Testing insertion operator overload. Bit array is inserted as a string into a stream and
  // then directly compared against its seed in string form to confirm both items are string type.

  FibLFSR InsertionTest("1111000011110000");
  std::stringstream testStream;
  testStream << InsertionTest;
  std::string testString;
  testStream >> testString;
  BOOST_REQUIRE(testString == "1111000011110000");
}
```

3 PS2: Dynamic N-Body Simulation

3.1 Objective

The objective of this assignment was to create an n-body particle simulation that would emulate the effects of n number of massive bodies and the gravitational forces that act upon each other. The assignment involved the creation of a planetary model that would represent our Sun and its four closest planets in our solar system and what their orbital patterns would look like based on the Newtonian gravitational forces acting upon each other.

3.2 Outcome

My implementation of the N-Body simulation worked out quite well. All planets included in our model stayed in a consistent orbit around the Sun and I was able to ensure this consistency through very long periods of time by adjusting the delta time variable used in the simulation to a very large figure.

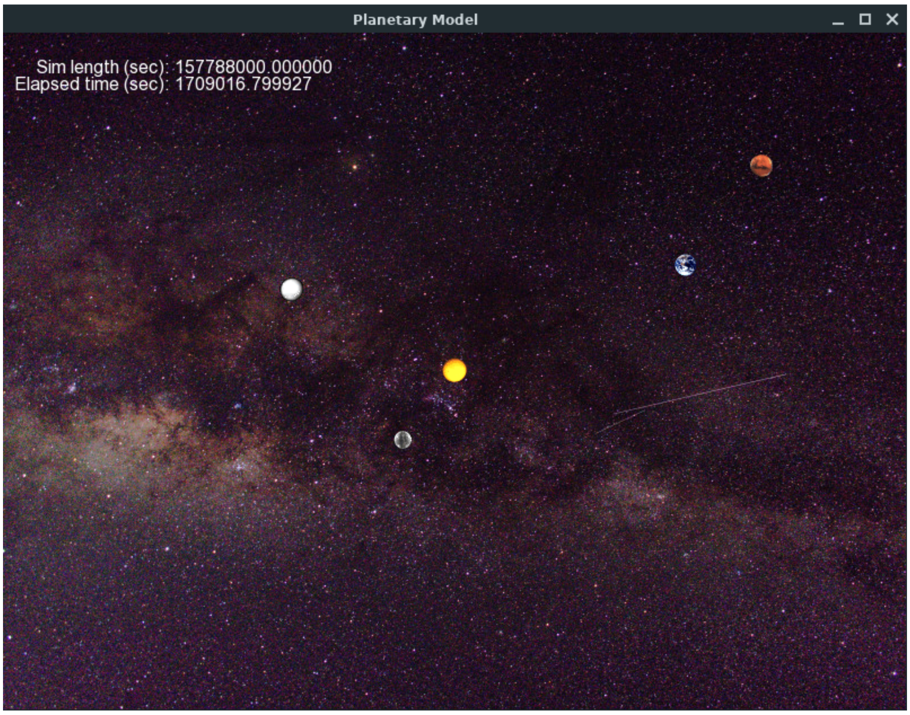

Figure 3.1: Screenshot of N-Body Simulation after running for an extended duration.

3.3 Implementation

The creation of the planetary model was done via the creation of a CelestialBody class that would represent each planetary body (or particle) in the simulation. The CelestialBody class contained all the required member fields and functions needed to determine its position on screen and the effect of other massive bodies against it. These member fields included the mass of the planet, three SFML objects (sf::Image, sf::Texture, sf::Sprite), and multiple vectors representing the physical location, velocity, net force, and screen position of each body. The member functions of the CelestialBody class included methods to calculate the physical force acting upon it from another body, to update its physical and on-screen location, and also to draw itself in the on screen window.

The ability for each CelestialBody to draw itself was done by inheriting from the sf::Drawable class and allowing the sf::RenderWindow object (our on-screen window) to call the friend function draw() for each CelestialBody object.

These CelestialBody objects were then stored in a larger Universe class object that would read from the command line, a variety of parameters needed to create the planetary model. These included the size of the universe, the duration of the simulation, and the delta time parameter for force calculation.

At execution, the N-Body program would be provided the universe parameters mentioned above as command line arguments from the user, and also would have input redirection taken from a text file containing all of the details for each planetary body in the simulation. At runtime, the Universe object is created using the passed command line arguments, and then the Universe object will then instantiate a CelestialBody object for each entry in the supplied input text file. Each CelestialBody object is stored in the Universe object after instantiation.

Once all bodies have been created and stored in the Universe object, an event loop is executed where the Universe object will increment the simulation by delta time seconds using its step() function once every frame. In this step() function, each CelestialBody stored in the Universe will proceed to calculate the net forces acting upon it for every other body by using it's CalcForce() method. Once every CelestialBody has calculated these forces; a new velocity, physical location, and on-screen location are then updated for each CelestialBody using the ApplyForce() method for each body. Lastly, the draw() function for each CelestialBody is called to update the on screen sprite representing each planet.

This process continues until our elapsed time has passed the simulation duration parameter passed by command line arguments at program execution.

3.4 Lessons Learned

The new concept that I learned in this assignment was the use of smart pointers. In C++ we do not have native garbage collection like in other languages such as Java, so it is up to the programmer to ensure they have appropriately cleaned up and freed their used memory, lest they end up with a memory leak. Smart pointers can help alleviate this by ensuring that the objects that are pointed to by each smart pointer are destroyed once the pointer itself has been destroyed or has gone out of scope.

Previous to this, I had been accomplishing dynamic memory allocation using the 'new' operator and ensuring the use of the 'delete' operator in the destructor of any object that required dynamic memory. In this assignment, we were required to use smart pointers to handle the destruction of our dynamically allocated CelestialBody objects. I opted to make use of the std::unique_ptr object to represent each body in the simulation and to have a vector of unique_prt's in my Universe object to store each CelestialBody pointer.

Once the simulation reached completion and the Universe object went out of scope, all unique pointers that pointed to each body were automatically freed along with each CelestialBody object itself. Thereby preventing any memory leaks from the dynamically allocated CelestialBody objects.

3.5 Code Listing

3.5.1 Makefile

```
CC = g++
CFLAGS = -Wall -Werror -pedantic --std=c++14
LIBS = -lsfml-graphics -lsfml-window -lsfml-system
DEPS = CelestialBody.hpp Universe.hpp

all: NBody

NBody: NBody.o CelestialBody.o Universe.o
	$(CC) $(CFLAGS) -o $@ $^ $(LIBS)

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

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

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

clean:
	rm *.o NBody
```

3.5.2 Driver Code

NBody.cpp

```
// Copyright 2022 - Ryan Casey
/**********************************************************
 *Name: Ryan Casey
 *Course name: <COMP.2040>
 *Assignment: PSX - Dynamic N-Body Simulation (Refactored and Linted)
 *Instructor's name: <Dr. James Daly>
 *Date: <3/3/2022>
 *Due Date: <3/14/2022>
 *Time to complete assignment: 2 hours.
```

```
 *Sources Of Help:

Linting guidelines:
https://google.github.io/styleguide/cppguide.html
************************************************************/
#include <iostream>
#include "CelestialBody.hpp"
#include "Universe.hpp"


using std::cin;

int main(int argc, char* argv[]) {
  // set viewport dimensions
  int viewportWidth = 800;
  int viewportHeight = 600;

  // Get simulation length and time delta from command line.
  double T = std::stod(argv[1]);
  double deltaTime = std::stod(argv[2]);

  // Create timing objects.
  sf::Clock clock;
  double elapsedTime;

  // Create text object to display currrent time.
  sf::Font font;
  font.loadFromFile("arial.ttf");
  sf::Text simLength;
  simLength.setFont(font);
  simLength.setCharacterSize(16);
  simLength.setStyle(sf::Text::Regular);
  simLength.setPosition(9, 20);

  // set text for current time of simulation
  std::string simLengthStr = std::to_string(T);
  simLengthStr = "      Sim length (sec): " + simLengthStr;
  simLength.setString(simLengthStr);

  // Create text object to display total simulation length.
  sf::Text currentTime;
  currentTime.setFont(font);
  currentTime.setCharacterSize(16);
  currentTime.setStyle(sf::Text::Regular);
  currentTime.setPosition(10, 35);

  // set text for total sim length.
  std::string elapsedTimeStr =
  std::to_string(clock.getElapsedTime().asSeconds() * deltaTime);
  elapsedTimeStr = "Elapsed time (sec): " + elapsedTimeStr;
  currentTime.setString(elapsedTimeStr);

  // Create render window object
  sf::RenderWindow window(sf::VideoMode(viewportWidth,
  viewportHeight), "Planetary Model");

  // Cap framerate to 60fps
  window.setFramerateLimit(60);
```

```

// Load background image.
sf::Image background;
background.loadFromFile("universe.jpg");
sf::Texture background_texture;
background_texture.loadFromImage(background);
sf::Sprite background_sprite;
background_sprite.setTexture(background_texture);

// Read in number of planets and radius of universe from stdin.
int numberOfBodies;
double universeRadius;
cin >> numberOfBodies >> universeRadius;

// create test universe
Universe testUniverse(viewportWidth, viewportHeight,
universeRadius, deltaTime);

// populate universe
testUniverse.add_bodies(numberOfBodies);

// Start window event loop.
while (window.isOpen() && elapsedTime <= T) {
  // Process events
  sf::Event event;
  while (window.pollEvent(event)) {
    // Close window if Close is clicked.
    if (event.type == sf::Event::Closed)
    window.close();
  }

  // Check for keyboard inputs
  if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape)) {
    // quit when 'Esc' is pressed...
    window.close();
  }

  // Clear the screen
  window.clear();

  // Draw background
  window.draw(background_sprite);

  // Draw bodies
  for (int i = 0; i < numberOfBodies; i++) {
    window.draw(testUniverse.GetBody(i));
  }

  // Take a step in deltaTime seconds.
  testUniverse.step(deltaTime);

  // Update time and draw to screen.
  elapsedTime = clock.getElapsedTime().asSeconds() * deltaTime;
  elapsedTimeStr =
  std::to_string(clock.getElapsedTime().asSeconds() * deltaTime);
  elapsedTimeStr = "Elapsed time (sec): " + elapsedTimeStr;
  currentTime.setString(elapsedTimeStr);
  window.draw(simLength);
```

```
    window.draw(currentTime);

    // Update window
    window.display();
  }

  // Print details of the universe to screen after sim ends.
  std::cout << numberOfBodies << std::endl << universeRadius << std::endl;
  testUniverse.PrintUniverseState();

  return 0;
}
```

3.5.3 Interface Files (.hpp)

CelestialBody.hpp

```
// Copyright 2022 - Ryan Casey
#pragma once
#include <math.h>
#include <string>
#include <iostream>
#include <memory>
#include <vector>
#include <SFML/Graphics.hpp>





class CelestialBody: public sf::Drawable {
 public:
    CelestialBody();
    ~CelestialBody();
    void GetBodyDetails();
    void SetUniverseParams(double universeRadius, int viewportWidth,
                           int viewportHeight, double deltaTime);
    void SetScreenParams();
    sf::Vector2<double> GetPosition() { return _position; }
    double GetMass() { return _mass; }
    void CalcForce(std::vector<std::unique_ptr<CelestialBody>>& otherBody);
    void ApplyForce();

    friend std::istream &operator>>(std::istream &input, CelestialBody &body);

 private:
    void draw(sf::RenderTarget& target,
              sf::RenderStates states) const override;
    sf::Vector2<double> _position;
    sf::Vector2<double> _velocity;
    sf::Vector2<double> _netForce;
    sf::Vector2<int> _viewportSize;
    sf::Vector2<int> _screenPos;
```

```

    double _mass;
    double _universeRadius;
    double _deltaTime;

    std::string _fileName;
    sf::Image _image;
    sf::Texture _texture;
    sf::Sprite _sprite;
};
```

Universe.hpp

```
// Copyright 2022 - Ryan Casey
#pragma once
#include <vector>
#include <iostream>
#include <memory>
#include "CelestialBody.hpp"

class Universe {
 public:
    Universe();
    Universe(int viewportWidth, int viewportHeight,
                double universeRadius, double deltaTime);
    void add_bodies(int numBodies);
    void PrintUniverseState();
    void step(double deltaTime);
    CelestialBody GetBody(int i);

 private:
    double _universeRadius;
    double _deltaTime;
    sf::Vector2<int> _viewportSize;
    std::vector<std::unique_ptr<CelestialBody>> _bodies;
};
```

3.5.4 Implementation Files (.cpp)

CelestialBody.cpp

```
// Copyright 2022 - Ryan Casey
#include "CelestialBody.hpp"

const double GRAV_FORCE = 6.67e-11;

CelestialBody::CelestialBody() {
    _position.x = 0;
    _position.y = 0;
    _velocity.x = 0;
    _velocity.y = 0;
    _netForce.x = 0;
    _netForce.y = 0;
```

```cpp
    _viewportSize.x = 0;
    _viewportSize.y = 0;
    _screenPos.x = 0;
    _screenPos.y = 0;
    _mass = 0;
    _universeRadius = 0;
    _deltaTime = 0;
    _fileName = "";
}
CelestialBody::~CelestialBody() {}

// Prints details on body.
void CelestialBody::GetBodyDetails() {
    std::cout <<
    _position.x << " " << _position.y << " " <<
    _velocity.x << " " <<  _velocity.y << " " <<
    _mass << " " << _fileName << std::endl;
}

//  Set screen position of body and load specified image file.
void CelestialBody::SetScreenParams() {
    // calc screen position
    _screenPos.x = (_universeRadius + _position.x) *
    (_viewportSize.x /(2 * _universeRadius));

    _screenPos.y = (_universeRadius - _position.y) *
    (_viewportSize.y /(2 * _universeRadius));

    // load sprite
    _image.loadFromFile(_fileName);
    _texture.loadFromImage(_image);
    _sprite.setTexture(_texture);

    // centers the origin of sprite to center of planet coordinates
    sf::Vector2u imageSize = _image.getSize();
    _sprite.setOrigin(imageSize.x / 2, imageSize.y / 2);

    // set final screen position
    _sprite.setPosition(_screenPos.x, _screenPos.y);
}

// Set universe details. Needed for screen position calculation.
void CelestialBody::SetUniverseParams(double universeRadius, int viewportWidth,
                                        int viewportHeight, double deltaTime) {
    _universeRadius = universeRadius;
    _viewportSize.x = viewportWidth;
    _viewportSize.y = viewportHeight;
    _deltaTime = deltaTime;
}

// Calc net force acting on the body,.
void CelestialBody::CalcForce(
                    std::vector<std::unique_ptr<CelestialBody>>& bodies) {
    double r;
    double force;
```

```cpp

    sf::Vector2<double> acceleration;
    sf::Vector2<double> posDelta, dirForce;


    for (auto& otherBody : bodies) {
        // skip body calculation if we are reading this body.
        if (_position.x == otherBody->_position.x &&
         _position.y == otherBody->_position.y) {
            continue;
        }

        // get distance 'r' between bodies.
        posDelta.x = (otherBody->_position.x - _position.x);
        posDelta.y = (otherBody->_position.y - _position.y);

        r = sqrt((posDelta.x * posDelta.x) + (posDelta.y * posDelta.y));

        // calc force generated
        force = (GRAV_FORCE * _mass * otherBody->_mass) / (r * r);
        dirForce.x = force * (posDelta.x / r);
        dirForce.y = force * (posDelta.y / r);

        // calc acceleration vector a = Force / mass
        acceleration.x = dirForce.x / _mass;
        acceleration.y = dirForce.y / _mass;

        // update velocity of body
        _velocity.x += acceleration.x * _deltaTime;
        _velocity.y += acceleration.y * _deltaTime;
    }
}

// Apply new velocity to body and update position.
void CelestialBody::ApplyForce() {
    // update new universe position
    _position.x += _velocity.x * _deltaTime;
    _position.y += _velocity.y * _deltaTime;

    // update screen position
    _screenPos.x =
    (_universeRadius + _position.x) * (_viewportSize.x /(2 * _universeRadius));
    _screenPos.y =
    (_universeRadius - _position.y) * (_viewportSize.y /(2 * _universeRadius));

    _sprite.setPosition(_screenPos.x, _screenPos.y);
}

// Draw function
void CelestialBody:: draw(sf::RenderTarget& target,
                              sf::RenderStates states) const {
    target.draw(_sprite);
}

// Extraction operator overload
std::istream &operator>>(std::istream &input, CelestialBody &body) {
```

```
    // read into member fields from input
    input >> body._position.x >> body._position.y >> body._velocity.x
    >> body._velocity.y >> body._mass >> body._fileName;

    return input;
}
```

Universe.cpp

```
// Copyright 2022 - Ryan Casey
#include "Universe.hpp"

Universe::Universe(int viewportWidth, int viewportHeight,
                    double universeRadius, double deltaTime) {
    _viewportSize.x = viewportWidth;
    _viewportSize.y = viewportHeight;
    _universeRadius = universeRadius;
    _deltaTime = deltaTime;
}

// Instantiate new CelestialBody object and read parameters from stdin.
// Push to Universe vector upon completion.
void Universe::add_bodies(int numBodies) {
    // Instantiate new CelestialBody, add to Universe vector.
    for (int i = 0; i < numBodies; i++) {
        _bodies.push_back(std::make_unique<CelestialBody>());
    }
    // Read in body details and set starting fields for each body.
    for (auto& body : _bodies) {
        std::cin >> *body;
        body->SetUniverseParams(_universeRadius, _viewportSize.x,
                                _viewportSize.y, _deltaTime);
        body->SetScreenParams();
    }
}

// Print details of the current state of the universe and its objects.
void Universe::PrintUniverseState() {
    for (auto& body : _bodies) {
        body->GetBodyDetails();
    }
}

// Progress the model of the unviverse by 'seconds' steps.
void Universe::step(double deltaTime) {
    // First, calc net force of all bodies acting on each other.
    for (auto& i : _bodies) {
        i->CalcForce(_bodies);
    }

    // Then apply forces to each body.
    for (auto& i : _bodies) {
        i->ApplyForce();
    }
}
// Get a body in universe.
```

```
CelestialBody Universe::GetBody(int i) {
    return *_bodies[i];
}
```

4 PS3: Triangle Fractal

4.1 Objective

The objective of this assignment was to create a class object named Triangle that would represent sub triangle in a larger Sierpinski fractal triangle. In addition to the class implementation we were also required to create a function fTree() for use in our driver code that would utilize our Triangle class to draw the Sierpinski triangle using the SFML library. Our fTree() function was also required to accept an integer argument to determine the depth of recursion for our drawing and a base length for the triangles to be drawn. These parameters for depth of recursion and triangle base length were passed to the driver code via command line arguments.

Along with the implementation of our Triangle class, we were also required to begin utilizing best practices regarding linting our code.

4.2 Outcome

I was successful in implementing the required functionality for this assignment as seen in Figure 4.1 below. In addition I was also able to add functionality that would randomly color the Sierpinski triangle a different triangle every time it was run. Likewise, I was also able to fully lint my code using the Google C++ code style guidelines.

One part of this assignment I felt I could improve on was my window scaling technique for larger Sierpinski triangles. As it stands right now, my SFML RenderWindow is setup to scale based upon an arbitrary triangle base length that I felt looked pleasing, but running the program with a large enough triangle base length does cause the RenderWindow to scale to dimensions that go off screen. This could be improved in the future by potentially clamping the RenderWindow size to the specified largest scale.

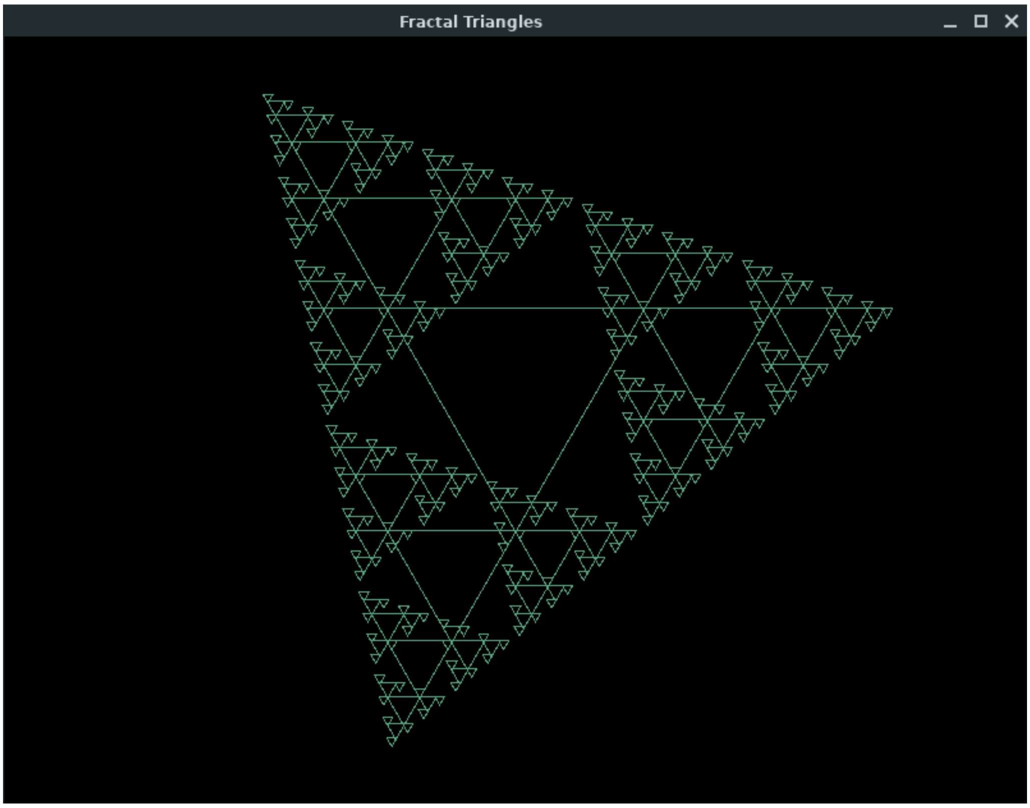

Figure 4.1: Output of TFractal program.

4.3 Implementation

My method of implementation for the Triangle class was done fairly simply by inheriting functionality from the SFML sf::ConvexShape class. This object is specialized for the use in drawing convex polygonal shapes and includes a variety of member functions that make it very apt for this assignment.

The member methods used to construct the Triangle were setPointCount() and setPoint(). These methods allowed for the designation of a specified number of vertices and the setting of each individual vertex with an index and sf::Vector position. As expected, 3 vertexes are set to represent a triangle.

In addition to the assignment of vertex positions, the member methods setOutlineColor(), setOutlineThickness(), and setFillColor() were used to specify the visual aspects of the triangles. In this case, they would be set to draw a thin outline with a transparent fill.

The actual drawing of the Sierpinski triangle was accomplished in the function fTree() in a recursive fashion. An initial base triangle is created using the passed base length, and three starting vertex positions. Once the base triangle is instantiated, three more recursive calls are made to fTree() to represent the 3 smaller sub triangles that were to be created at each of the vertexes of the larger base triangle.

These calls fTree() that create the sub-triangles are passed a base length that is 1/2 of the original base length. Each of the sub calls to fTree() borrows one vertex from the previous larger triangle as a reference point for one of its own vertexes. The position of the other two vertexes of the sub-triangles are determined using some simple geometry pertaining to equilateral right triangles. In effect ever sub-triangle will "borrow" one vertex from its parent and determine the other two on its own.

These recursive calls for sub-triangles are continued until we reach the desired depth of recursion. After this, each Triangle object is draw on screen to the SFML RenderWindow each frame using the draw() method that Triangle inherited from sf::ConvexShape.

Scaling of the view port to accommodate larger triangle base lengths was accomplished by determining a triangle base length that seemed to best fit the default dimensions I chose for the RenderWindow and then creating a scaling ratio if a triangle base length passed to the program was larger than my chosen size. This ratio was then applied to the view port dimensions of the RenderWindow.

The final touch I added to this program to make it more visually appealing was to randomly select a set of RGB values and to pass those to the Triangle objects being created. This had the effect of always giving a different color triangle every time the program is ran.

4.4 Lessons Learned

The core concepts used in my program were mainly the use of recursion and inheritance of existing SFML objects that were best suited to the task. While these concepts regarding implementation were not new to me, the aspect that was foreign to me at the start was the concept of code linting.

Prior to this assignment we were introduced to the idea of linting as a means to create more readable and uniform code. We were encouraged to utilize the Google C++ code style guidelines and were also required to run our code against a lint file that would ensure the Google linting guidelines were being followed in our code.

Becoming accustomed to linting my own code was beneficial to me as it taught me what "clean code" should look like and showed me many techniques that are considered best practice in regards to structuring my code. This is important as in the future, we would want our code to be easily readable not only for ourselves but also for others. All further assignments from this one feature fully linted code that is much easier to digest as human reader.

4.5 Code Listing

4.5.1 Makefile

```
CC = g++
CFLAGS = -Wall -Werror -pedantic --std=c++14
LIBS = -lsfml-graphics -lsfml-window -lsfml-system
DEPS = Triangle.hpp

all: TFractal

TFractal: TFractal.o Triangle.o
	$(CC) $(CFLAGS) -o $@ $^ $(LIBS)

```

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

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

clean:
	rm *.o TFractal
```

4.5.2 Driver Code

TFractal.cpp

```cpp
// Copyright 2022 <Ryan Casey>
/**********************************************************
 *Name: Ryan Casey
 *Course name: <COMP.2040>
 *Assignment: PS3 - Triangle Fractal
 *Instructor's name: <Dr. James Daly>
 *Date: <2/28/2022>
 *Due Date: <2/28/2022>
 *Time to complete assignment: 4 hours.
 *Sources Of Help:

https://www.sfml-dev.org/documentation/2.5.1/classsf_1_1ConvexShape.php#a6598feed5fea1325a36b0f3a615ac55c
https://google.github.io/styleguide/cppguide.html

**********************************************************/

#include <time.h>
#include <math.h>

#include <iostream>
#include <string>

#include "Triangle.hpp"


void fTree(sf::Color color, float base,
    sf::Vector2f left, sf::Vector2f right, sf::Vector2f bottom,
    int iterations, sf::RenderTarget& window);

int main(int argc, char* argv[]) {
    // set viewport dimensions
    int viewportWidth = 800;
    int viewportHeight = 600;

    // Set fractal parameters from cmd line args
    float triangleBase = std::stof(argv[1]);
    float triangleHeight = triangleBase * (sqrt(3) / 2);
    int fractalDepth = std::stoi(argv[2]);

    // Scale SFML window dimensions if triangle base gets too large.
    if (triangleBase > 225) {
        float ratio = triangleBase / 225;
```

```
        viewportWidth *= ratio;
        viewportHeight *= ratio;
    }

    // Set screen midpoint and initial vertex positions for first triangle
    sf::Vector2f screenMidPoint(viewportWidth/2, viewportHeight/2);
    sf::Vector2f bottom(screenMidPoint.x, screenMidPoint.y + triangleHeight/2);
    sf::Vector2f left(bottom.x - (triangleBase/2), bottom.y - triangleHeight);
    sf::Vector2f right(bottom.x + (triangleBase/2), bottom.y - triangleHeight);

    // Create window object
    sf::RenderWindow window(sf::VideoMode(viewportWidth, viewportHeight),
    "Fractal Triangles");

    // Cap framerate to 60fps
    window.setFramerateLimit(60);

    // set seed and pick random color for triangles.
    unsigned int seed = time(NULL);
    sf::Color randColor(rand_r(&seed)%255,
    rand_r(&seed)%255,
    rand_r(&seed)%255, 255);

    // Start Game Loop
    while (window.isOpen()) {
        // Process events
        sf::Event event;
        while (window.pollEvent(event)) {
            // Close window: exit
            if (event.type == sf::Event::Closed)
            window.close();
            }

            // Check for keyboard inputs
            if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape))
            // quit when 'Esc' is pressed...
            window.close();

            // Clear the screen
            window.clear();

            // Draw here
            fTree(randColor, triangleBase, left,
            right, bottom, fractalDepth, window);

            // Update window
            window.display();
            }

    return 0;
}

/* Recursively draw triangles.
Params:
```

```
    color - color of trangle.
    base - length of side of triangle.
    left, right, bottom - initial positions of triangle vertexes.
    iterations - recursion depth.
    window - RenderTarget reference to draw triangles to.
*/
void fTree(sf::Color color, float base,
sf::Vector2f left, sf::Vector2f right, sf::Vector2f bottom,
int iterations, sf::RenderTarget& window) {
    // set params for triangle
    float triBase = base;
    float triHeight = triBase * (sqrt(3) / 2);

    // create and draw this triangle
    Triangle tri(left, right, bottom);
    tri.setOutlineColor(color);
    window.draw(tri);

    // once we've hit the final depth of iterations, recurse back up.
    if (iterations == 0) {
        return;
    }

    // bifuricate length of triangle base for following triangles.
    triBase /= 2;
    triHeight = triBase * (sqrt(3) / 2);

    // recursive calls to smaller triangles.
    fTree(color, triBase,
        sf::Vector2f(left.x - triBase /2, left.y - triHeight),
        sf::Vector2f(left.x + triBase /2, left.y - triHeight),
        left,
        iterations-1, window);

    fTree(color, triBase,
        right,
        sf::Vector2f(right.x + triBase, right.y),
        sf::Vector2f(right.x + triBase /2, right.y + triHeight),
        iterations-1, window);

    fTree(color, triBase,
        sf::Vector2f(bottom.x - triBase, bottom.y),
        bottom,
        sf::Vector2f(bottom.x - triBase /2, bottom.y + triHeight),
        iterations-1, window);
}
```

4.5.3 Interface Files (.hpp)

Triangle.hpp

```
// Copyright 2022 <Ryan Casey>
#pragma once
#include <SFML/Graphics.hpp>

class Triangle: public sf::ConvexShape {
```

```
public:
    Triangle();
    Triangle(sf::Vector2f left, sf::Vector2f right, sf::Vector2f bottom);
    virtual ~Triangle() {}

private:
};
```

4.5.4 Implementation Files (.cpp)

Triangle.cpp

```
// Copyright 2022 <Ryan Casey>
#include "Triangle.hpp"

Triangle::Triangle() {
    this->setPointCount(3);
    this->setOutlineColor(sf::Color::White);
    this->setFillColor(sf::Color::Transparent);
    this->setOutlineThickness(1);
}

Triangle::Triangle(sf::Vector2f left, sf::Vector2f right, sf::Vector2f ↩
  bottom) {
    this->setPointCount(3);
    this->setOutlineColor(sf::Color::White);
    this->setOutlineThickness(1);
    this->setFillColor(sf::Color::Transparent);
    this->setPoint(0, left);
    this->setPoint(1, right);
    this->setPoint(2, bottom);
}
```

5 PS4: StringSound

5.1 Objective

The objective of this assignment was to utilize the SFML Audio library to create a class StringSound that would emulate the sound of a guitar string being plucked. Additionally, we were also required to implement a virtual keyboard that would play different notes from our StringSound class based on user keystrokes.

5.2 Outcome

This assignment turned out well for me as I was able to implement all required functionality. Additionally I was able to get my program to play multiple sounds at once as to emulate guitar chords.

One area I think could be improved upon however was the pitch of my guitar string sound. Using the default implementation requirements, the guitar sound is rather high pitched. With some more adjustment I believe I could improve the quality of the sound to be closer to that of a real guitar or harpsichord by changing the formula used for pitch calculation.

Seen in Figure 5.1 is the terminal output of the keystrokes for the virtual keyboard. As you can see, all 37 keyboard keys are struck based upon a designated key stroke.

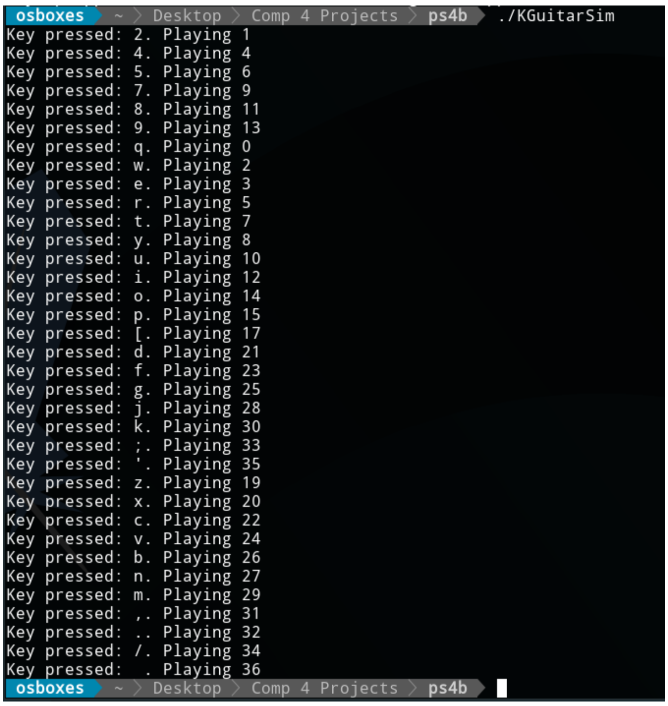

Figure 5.1: Terminal output of PS4 StringSound.

5.3 Implementation

My program was implemented in three parts: the StringSound class, the virtual keyboard, and corresponding map of keys. My StringSound class was implemented by the use of a std::deque to emulate a circular buffer. This circular buffer would be used to emulate the sound of a string by first being filled with random values and then having a decay function applied to the first value of the buffer with that new value being subsequently added to the end of the buffer. The capacity of the circular buffer created by the StringSound class is determined at instantiation of the StringSound based upon the frequency parameter passed to it.

The act of populating the StringSound buffer with random values was done via the pluck() function and as its name indicates, this function would emulate the random waveform noise that is created when a string is plucked. Using the previously mentioned decay function, an averaging of these buffer values would emulate the string coming back to a resting state. This process is accomplished in an incremental step by step fashion using the tic() function in my StringSound class. The first value of the buffer is then accessible by use of the sample() function.

To implement the keyboard and all 37 notes, 37 StringSounds were created at runtime and each StringSound's sample() function was used to populate a separate vector of samples. These 37 vectors of samples were then stored in their own vector called **vecSamples**. Using the SFML sf::SoundBuffer object, each item in **vecSamples** was stored into its own sf::SoundBuffer. These 37 sf::SoundBuffers were stored into their own vector in a similar fashion as before, called **vecSoundBuffers**. Lastly, the same process was repeated again but this time passing each of the 37 sf::SoundBuffer objects into its own sf::Sound object. Again, these were stored in their own respective vector, **vecSounds**.

It should also be noted that as each StringSound object is created, the frequency of the note it represents was passed to it as a constructor parameter and that the objects themselves were then quickly discarded once the samples had been extracted from them into **vecSamples**. Once these steps had been accomplished, each of the 37 sf::Sound objects was then stored in a map so they could be accessed later.

The final step in my implementation was mapping the user keystrokes to each of the notes. The reading of user keystrokes was accomplished using an event loop as seen in previous assignments. The event sf::Event::KeyPressed was checked for on every frame as the program ran to see if the user had made an input.

To determine which note should be played when a key is inputted, I created a class called KeyMap which contained two maps. One map, called **_keymap**, would map the key code returned by the sf::Event::KeyPressed event to its respective char value. The second map, called **_notemap**, would map each char value to its corresponding index in **vecSounds**. This mapping would allow a key stroke from the user to be mapped to the appropriate note and then be played using the sf::Sound member function play().

5.4 Lessons Learned

The main concept that I took away from this assignment was the idea of a ring or circular buffer. A circular buffer is a data structure that utilizes a buffer that allows the pushing of items onto it from one side up to a maximum capacity. The circular buffer will process items in first in, first out order and then can discard processed elements or simply wait for new items to be pushed on, causing older items to be popped off.

The use of a circular buffer can be very useful in situations where data is being received and queued in an asynchronous fashion such as a data stream. The buffer will process the data in order as it arrives and will make room for new items as quickly as processing will allow.

5.5 Code Listing

5.5.1 Makefile

```
CC = g++
CFLAGS = -Wall -Werror -pedantic --std=c++14
DEPS = CircularBuffer.hpp StringSound.hpp keymap.hpp
LIBS = -lboost_unit_test_framework -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio
SOURCES = CircularBuffer.cpp StringSound.cpp KGuitarSim.cpp test.cpp
LINT = /usr/local/bin/cpplint.py
LINTFLAGS = --filter=-runtime/references,-build/header_guard,-build/c++11 --extensions=cpp,hpp

all: KGuitarSim

KGuitarSim: KGuitarSim.o CircularBuffer.o StringSound.o keymap.o
	$(CC) $(CFLAGS) -o $@ $^ $(LIBS)

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

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

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

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

test: test.o CircularBuffer.o StringSound.o
	$(CC) $(CFLAGS) -o $@ $^ $(LIBS)

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

lint: $(SOURCES) $(DEPS)
	$(LINT) $(LINTFLAGS) $(SOURCES) $(DEPS)

clean:
	rm *.o KGuitarSim test
```

5.5.2 Driver Code

KGuitarSim.cpp

```
// Copyright 2022 - Ryan Casey
/**
* PS4B - StringSound
* Instructor: James Daly
* Todays Date: 3/28/2022
* Due Date : 3/28/2022
* Program Description:
*
* In this program I have implemented a of a StringSound
* class that emulates the vibration of a guitar string using
* our previously created CircularBuffer class from ps4a.
*
* I have also implemented for these guitar sounds to be played
* by utilizing my StringSound object in conjunction with
* audio resource classes provided in the SFML audio resource library.
*
* Lastly, I also created the implementation of a virtual
* "piano" keyboard that will play the approriate guitar note
* when the corresponding key of a provided keyboard layout
* is pressed. This was achieved by converting sfml keyboard
* events into inputs that I could map to the appropriate note.
*
* Created by: Ryan Casey
**/
#include <math.h>

#include <iostream>
#include <map>
#include <memory>

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

#include "StringSound.hpp"
#include "keymap.hpp"

#define SAMPLE_RATE 44100
#define NUM_NOTES 37

std::vector<sf::Int16> makeSampleVec(StringSound& stringBuffer);

int main() {
  // Create window and event objs.
  sf::RenderWindow window(sf::VideoMode(300, 200), "SFML Keyboard!");
  sf::Event event;

  // vectors to hold samples, SoundBuffers, and Sounds
  std::vector<std::vector<sf::Int16>> vecSamples;
  std::vector<std::shared_ptr<sf::SoundBuffer>> vecSoundBuffers;
  std::vector<sf::Sound> vecSounds;

```

```cpp
// lambda function used to calc correct frequency of notes.
auto calcFreq = [](int i) { return 440.0 * pow(2, ((i - 24.0) / 12.0)); };

// Load StringSound samples to vecSamples
for (size_t i = 0; i < NUM_NOTES; i++) {
  double freq = ceil(calcFreq(i));
  std::vector<sf::Int16> samples;

  StringSound tempString(freq);
  samples = makeSampleVec(tempString);
  vecSamples.push_back(samples);
}

// Load sf::SoundBuffers to vecSoundBuffers
for (size_t i = 0; i < NUM_NOTES; i++) {
  std::shared_ptr<sf::SoundBuffer> sBuff =
  std::make_shared<sf::SoundBuffer> ();

  // Load samples to SoundBuffer
  sBuff->loadFromSamples(&vecSamples[i][0],
  vecSamples.at(i).size(), 2, SAMPLE_RATE);

  vecSoundBuffers.push_back(sBuff);
}

// Load sf::Sounds to vecSounds
for (size_t i = 0; i < NUM_NOTES; i++) {
  sf::Sound sound;
  sound.setBuffer(*vecSoundBuffers[i]);
  vecSounds.push_back(sound);
}

// map for key press event codes.
KeyMap keyMap;
std::string keyboardLayout = "q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,.;/' ";
char keyPressed;
size_t noteIndex;
size_t keyFound;


while (window.isOpen()) {
  while (window.pollEvent(event)) {
    switch (event.type) {
      case sf::Event::Closed:
        window.close();
        break;

      case sf::Event::KeyPressed:
        // Get correct note based on key press.
        keyPressed = keyMap.getKey(event.key.code);
        noteIndex = keyMap.getNote(keyPressed);
        // Play note if key press is included in keyboard.
        keyFound = keyboardLayout.find(keyPressed);
        if (keyFound != std::string::npos) {
          vecSounds.at(noteIndex).play();
        }

```

```
      default:
        break;
    }
    window.clear();
    window.display();
  }
}

return 0;
}

// Fill a vector with samples from StringSound object.
std::vector<sf::Int16> makeSampleVec(StringSound& stringBuffer) {
  // Create a vector to fill.
  std::vector<sf::Int16> samples;
  // Populate buffer
  stringBuffer.pluck();
  // Duration of buffer in seconds.
  int duration = 8;


  for (int i = 0; i < SAMPLE_RATE * duration; i++) {
    // Progress buffer by one tic.
    stringBuffer.tic();
    // Push first sample onto vector.
    samples.push_back(stringBuffer.sample());
  }
  return samples;
}
```

5.5.3 Interface Files (.hpp)

CircularBuffer.hpp

```
// Copyright 2022 - Ryan Casey
#pragma once
#include <stdint.h>
#include <deque>

class CircularBuffer {
 public:
    explicit CircularBuffer(size_t capacity);

    size_t size();
    size_t capacity();
    bool isEmpty();
    bool isFull();
    void enqueue(int16_t x);
    int16_t dequeue();
    int16_t peek();
    void clear() { _buffer.clear(); }
    void print();

 private:
    std::deque<int16_t> _buffer;
    size_t _capacity;
```

```
};
```

StringSound.hpp

```
// Copyright 2022 - Ryan Casey
#pragma once
#include <vector>
#include <SFML/Audio.hpp>
#include "CircularBuffer.hpp"

class StringSound {
 public:
  explicit StringSound(double frequency);
  explicit StringSound(std::vector<sf::Int16> init);
  StringSound(const StringSound& obj) = delete;  // copy constructer deleted.
  ~StringSound();
  void pluck();
  void tic();
  sf::Int16 sample() { return _cb->peek(); }
  int time() { return _time; }

 private:
  CircularBuffer* _cb;
  int _time;
  unsigned int _seed;
};
```

keymap.hpp

```
// Copyright 2022 - Ryan Casey
#pragma once
#include <map>
#include <string>

class KeyMap {
 public:
  KeyMap();
  char getKey(int i) { return _keymap[i]; }
  int getNote(char ch) { return _notemap[ch]; }
 private:
  std::map<int, char> _keymap;
  std::map<char, int> _notemap;
  std::string keyLayout = "q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,.;/' ";
};
```

5.5.4 Implementation Files (.cpp)

CircularBuffer.cpp

```
// Copyright 2022 - Ryan Casey
```

```
#include <stdexcept>
#include <iostream>
#include "CircularBuffer.hpp"

// Value constructor sets the capacity of the ring buffer
// and ensures it is at least > 1.
CircularBuffer::CircularBuffer(size_t capacity) {
  // Throw when capacity is less than 1.
  if (static_cast<int>(capacity) < 1) {
    throw std::invalid_argument(
    "CircularBuffer constructor: capacity must be greater than 0\n");
  } else {
    // Otherwise set capacity of buffer
    _capacity = capacity;
  }
}
// Return capacity of buffer.
size_t CircularBuffer::capacity() {
  return _capacity;
}

// Return number of elements in buffer.
size_t CircularBuffer::size() {
  return _buffer.size();
}

// Return whether buffer is empty.
bool CircularBuffer::isEmpty() {
  return (_buffer.empty());
}

// Return whether buffer is at capacity.
bool CircularBuffer::isFull() {
  return _buffer.size() >= _capacity;
}

// Add an element to the end of the buffer.
void CircularBuffer::enqueue(int16_t i) {
  if (isFull()) {
    throw std::runtime_error("enqueue: can't enqueue to a full ring\n");
  } else {
    _buffer.push_back(i);
  }
}

// Store and then pop the element at front of buffer. Returns value ↩
    popped.
int16_t CircularBuffer::dequeue() {
  if (_buffer.empty()) {
    throw(std::runtime_error("dequeue: cannot dequeue from empty ring.\n"↩
        ));
  } else {
    int16_t returnVal = _buffer.front();
    _buffer.pop_front();
    return returnVal;
  }
}

```

```
// Returns value at front of buffer.
int16_t CircularBuffer::peek() {
  if (_buffer.empty()) {
    throw(std::runtime_error("peek: cannot peek into an empty buffer.\n"));
  } else {
    return _buffer.front();
  }
}

// Print buffer contents.
void CircularBuffer::print() {
  for (auto i : _buffer) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
}
```

StringSound.cpp

```
// Copyright 2022 - Ryan Casey
#include <math.h>

#include <iostream>
#include <random>
#include <chrono>
#include <exception>

#include "StringSound.hpp"

#define SAMPLE_RATE 44100

// Create a guitar string sound of the given
// frequency using a sampling rate of 44,100.
StringSound::StringSound(double frequency) {
  if (frequency <= 0)
    throw(std::invalid_argument(
    "StringSound: Cannot create StringSound use a frequency <= 0."));

  // Size of buffer = SAMPLE_RATE / ceil(frequency)
  double buffSize = ceil(SAMPLE_RATE / frequency);

  _cb = new CircularBuffer(buffSize);

  // Set seed for rand val generation and init time.
  _seed = std::chrono::system_clock::now().time_since_epoch().count();
  _time = 0;
}
// Create string sound that can accomodate the passed vector.
StringSound::StringSound(std::vector<sf::Int16> init) {
  if (init.size() == 0)
    throw(std::invalid_argument(
    "StringSound: Cannot create StringSound using empty sample vector."));

  // Create vec that is size of init and push on elements from init.
```

```
  _cb = new CircularBuffer(init.size());

  for (auto i : init) {
    _cb->enqueue(i);
  }

  // Set seed for rand val generation.
  _seed = std::chrono::system_clock::now().time_since_epoch().count();
  _time = 0;
}

StringSound::~StringSound() {
  // delete CircleBuffer object.
  delete _cb;
}

// Pluck the guitar string by replacing the
// buffer with random values,representing white noise.
void StringSound::pluck() {
  // Clear the existing CircularBuffer.
  _cb->clear();

  // Enqueue new random values up to capacity of buffer.
  std::mt19937 randVal(_seed);
  for (size_t i = 0; i < _cb->capacity(); i++) {
    _cb->enqueue(static_cast<int16_t>(randVal()));
  }
}

// Advance the simulation of the string by one time step.
void StringSound::tic() {
  // Compute on first two elements in buffer.
  // Store first sample then pop first sample.
  int16_t frontVal = _cb->dequeue();

  // Calc new value to add to end of buffer.
  int16_t newSample = (frontVal + this->sample()) * 0.5 * 0.996;
  _cb->enqueue(newSample);

  // incrememnt time
  _time++;
}
```

keymap.cpp

```
// Copyright 2022- Ryan Casey
#include "keymap.hpp"

KeyMap::KeyMap() {
  // Map sf::Keyboard event codes to appropriate char values.
  _keymap[16] = 'q';
  _keymap[28] = '2';
  _keymap[22] = 'w';
  _keymap[4] = 'e';
  _keymap[30] = '4';
  _keymap[17] = 'r';
```

```
  _keymap[31] = '5';
  _keymap[19] = 't';
  _keymap[24] = 'y';
  _keymap[33] = '7';
  _keymap[20] = 'u';
  _keymap[34] = '8';
  _keymap[8] = 'i';
  _keymap[35] = '9';
  _keymap[14] = 'o';
  _keymap[15] = 'p';
  _keymap[56] = '-';
  _keymap[46] = '[';
  _keymap[55] = '=';
  _keymap[25] = 'z';
  _keymap[23] = 'x';
  _keymap[3] = 'd';
  _keymap[2] = 'c';
  _keymap[5] = 'f';
  _keymap[21] = 'v';
  _keymap[6] = 'g';
  _keymap[1] = 'b';
  _keymap[13] = 'n';
  _keymap[9] = 'j';
  _keymap[12] = 'm';
  _keymap[10] = 'k';
  _keymap[49] = ',';
  _keymap[50] = '.';
  _keymap[48] = ';';
  _keymap[52] = '/';
  _keymap[51] = '\'';
  _keymap[57] = ' ';

  // Map char values to index values.
  int i = 0;
  for(char ch: keyLayout) {
    _notemap[ch] = i;
    i++;
  }
}
```

6 PS5: DNA Alignment

6.1 Objective

The objective of this assignment was to implement a class called EDistance that would compute the optimal edit ”distance” between two misaligned strings and to determine the optimal way in which they can be aligned. The alignment of the two strings is to be based upon the optimal combination of matching characters, non-matching characters, and addition of gaps to achieve equal length between the two strings.

Once optimal alignment of the strings had been achieved, the execution time of the program was to be outputted to the terminal. Along with these requirements, we were also tasked with ensuring our program gracefully handled the allocation and deallocation of dynamic memory at run time.

6.2 Outcome

I was successful in meeting the requirements for this assignment and was pleasantly surprised how well the program could tackle strings of increasingly longer length. A slew of sample strings were redirected to the program to ensure the was correctly calculating the optimal distance between the two.

One part of this assignment that I did want to improve was the run time of the program when dealing with strings of a much longer magnitude, such as DNA sequences. I believe one reason why my program seemed to falter at longer string lengths however was due to the resource limitations of my development environment. For this class I have been developing my code using a virtual machine which is limited to 1GB of system memory. I believe the fact that my CPU is doing double duty to run both Windows and Ubuntu definitely hampered the execution time for longer strings and the fact that my virtual environment is only allocated 1GB of system memory meant that my program would typically run out of accessible memory fairly quickly.

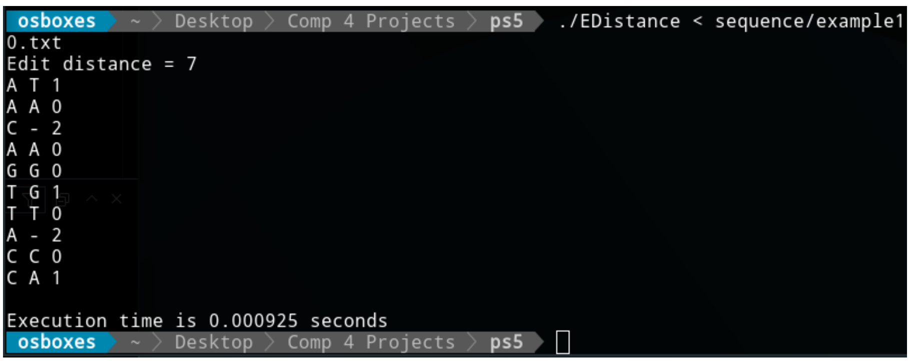

Figure 6.1: Results of string alignment in PS5.

6.3 Implementation

In this assignment we were provided with multiple potential algorithms to use to accomplish this task. I elected to go about this tasking using the Needleman and Wunsch algorithm. To implement this algorithm, my EDistance class used a 2 dimensional array that was (N+1)x(M+1) in length where N and M represent the length of the strings being aligned.

Using the optDistance() method, the furthest furthest column and bottom row of the array are filled with multiples of our distance "cost" for inserting a gap into our string. This value is commonly called the *indel* value. These starting values are needed in order to begin the process of comparing the chars of each string and determining if a gap may be the optimal choice in either string.

Then starting from the bottom-right most unpopulated position of the array, optDistance() will traverse the array from right to left, bottom to top. At each cell in the array, the character at the "row" position of our first sting is compared to the character at the "column" position of the second string. For example, if we are in at array index array[0][1], then the character at position 0 in the first sting is compared to the character at position 1 in the second string.

At each comparison step, three cases are considered. If the characters match, their cost is considered 0. If the characters do no match, their cost is considered 1. If a character is matched with a gap its cost is considered 2. The costs associated with all 3 possibilities are calculated using the current comparison cost and the previously calculated costs of the surrounding cells that are immediately to the right, to the bottom, or to the bottom right of the current one being inspected. Once every cell of the array has been populated using the costs of every possible matching condition we can then inspect the first cell of the of the array **_distMatrix**[0][0]. This final cell will represent the optimal "distance" cost associated with aligning the strings. This optimal distance is then returned by optDistance().

Once the cost matrix **_distMatrix** has been populated by optDistance(), it is then traced backwards by the function alignment(). The alignment() function will start from **_distMatrix**[0][0] and inspect the costs of the cells that surround it. Again these considered cells are directly below it, to the right, or to the bottom right of it in the matrix. Of these three inspected cells, alignment() will select the cheapest cost option as the "optimal" choice and then will assemble two separate strings char by char based upon which direction the optimal choice was towards.

If the optimal choice was taken from the cell to the right, then a gap is added to our first string while the second string has its respective character placed. If the cell below was taken, then a gap is added to the second string while the first string gets its own character. Lastly if the cell to the bottom right was chosen, then both strings get their respective characters placed.

Once the two resulting strings have been built, they are combined into one result string which is then returned in a formatted vertical format with the edit cost of each comparison listed next to the characters at that step.

The last step in this program was outputting the duration of the program execution time. This was done fairly easily using the SFML class' sf::Clock and sf::Time.

6.4 Lessons Learned

The main concepts that I took from this assignment were the use of dynamic programming, and memory management. Dynamic programming is the idea of solving a larger optimization problem using the results of smaller more manageable problems. In this instance we determined the optimal string alignment by first using an algorithm to determine and populate a cost matrix that reflected all possible comparison costs for either string. Then using a second algorithm, we traversed our results to find the optimal choices at each step in the matrix and used that to create the optimally aligned strings.

In regards to memory management, this assignment did clearly show me the impact of space and time complexity when dealing will problems at larger scales. While I was able to ensure that my program was free of memory leaks, I was not able to overcome the resource restrictions of my computer.

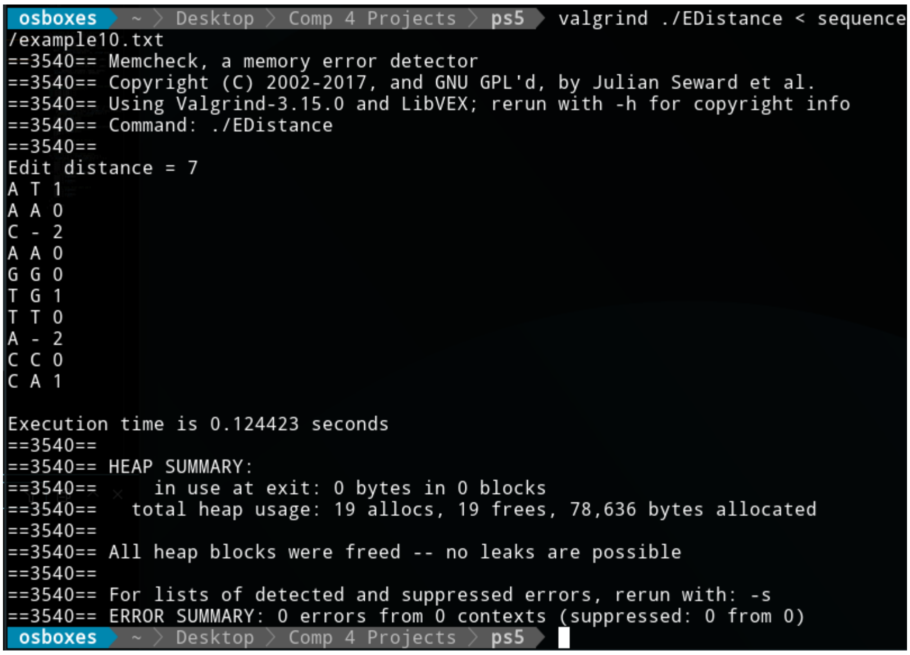

Figure 6.2: Running Valgrind on PS5 to ensure no leaks.

My use of a two dimensional array created a space complexity of N^2 that put me in a situation where the doubling of run time caused a quadrupling of space requirements to run the program. So in effect, as the strings doubled in length, they required four times as much memory to occupy the matrix. This did lead me to realize that there were more optimal solutions in terms of space complexity, such as using the Hirschberg algorithm for creating a two dimensional array in a linear fashion.

6.5 Code Listing

6.5.1 Makefile

```
CC = g++ -g
CFLAGS = -Wall -Werror -pedantic --std=c++14
DEPS = EDistance.hpp
LIBS = -lboost_unit_test_framework -lsfml-system
SOURCES = main.cpp EDistance.cpp
LINT = /usr/local/bin/cpplint.py
LINTFLAGS = --filter=-runtime/references,-build/header_guard --extensions=cpp,hpp

all: EDistance

EDistance: main.o EDistance.o
	$(CC) $(CFLAGS) -o $@ $^ $(LIBS)

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

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

lint: $(SOURCES) $(DEPS)
	$(LINT) $(LINTFLAGS) $(SOURCES) $(DEPS)

clean:
	rm *.o EDistance
```

6.5.2 Driver Code

main.cpp

```cpp
// Copyright 2022 - Ryan Casey
#include <iostream>
#include <SFML/System.hpp>
#include "EDistance.hpp"


int main() {
  sf::Clock clock;
  sf::Time t;
  std::string str1, str2;
  std::cin >> str1;
  std::cin >> str2;

  EDistance ed(str1, str2);

  std::cout << "Edit distance = " << ed.optDistance() << std::endl;
  std::cout << ed.alignment() << std::endl;

  t = clock.getElapsedTime();
  std::cout << "Execution time is " << t.asSeconds() << " seconds\n";
```

```
  return 0;
}
```

6.5.3 Interface Files (.hpp)

EDistance.hpp

```
// Copyright 2022 - Ryan Casey
#include <string>
#include <vector>
#include <memory>
#include <algorithm>
class EDistance {
 public:
  EDistance(std::string x, std::string y);
  ~EDistance();
  static int penalty(char a, char b);
  static int min(int a, int b, int c);
  int optDistance();
  std::string alignment();
  void print();
 private:
  std::vector<std::vector<int>>* _distMatrix;
  std::string _stringX, _stringY;
  size_t _rowCount;
  size_t _colCount;
  int _inDel = 2;
};
```

6.5.4 Implementation Files (.cpp)

EDistance.cpp

```
// Copyright 2022 - Ryan Casey
/**
* PS4B - StringSound
* Instructor: James Daly
* Todays Date: 3/28/2022
* Due Date : 3/28/2022
* Time to complete: 13 hours
* Program Description:

In this project I have implemented a string sequence
aligner that is constructed using a two dimensonal
array. The method for populating and retracing my
distance matrix was done using the N&W Algortihm.

With that said, the optimal edit distance for the
two supplied strings is located at postion (0,0) in
my distance matrix and the value of which is returned
by the optDitance() function.

Once the matrix has been populated with possible optimal
```

```cpp
options. Using the alignment() function, my matrix will
be traced backwards starting from point (0,0) to determine
what the optimal choice was at each step along the way.
Using these retraced steps, a new string
was created which appropriately matched the two strings
together considering any gaps that need to be added/removed
and then prints a formated vertical version of both strings
and the associated penalty for each char of each string.

* Created by: Ryan Casey
**/

#include <memory>
#include <iostream>
#include <algorithm>
#include <iomanip>
#include <exception>
#include "EDistance.hpp"

EDistance::EDistance(std::string x, std::string y) {
  if (x.length() < 1 || y.length() < 1) {
    throw std::invalid_argument("Exception: String inputs must be greater than 0.");
  }
  _stringX = x;
  _stringY = y;

  _rowCount = _stringX.length() + 1;
  _colCount = _stringY.length() + 1;

  _distMatrix = new std::vector<std::vector<int>>(_rowCount,
  std::vector<int>(_colCount, 0));
}

EDistance::~EDistance() {
  delete _distMatrix;
}

// Return the penalty for mismatched chars.
int EDistance::penalty(char a, char b) {
  if (a == b) {
    return 0;
  }
  return 1;
}

// Return min value.
int EDistance::min(int a, int b, int c) {
  int min = std::min(a, b);
  min = std::min(min, c);
  return min;
}

// Create and populate distance matrix based on strings.
int EDistance::optDistance() {
  size_t match, remove, insert;

  // Fill furthest column bottom to top w/ multiples of inDel.
```

```
  for (size_t i = 0; i < _rowCount; i++) {
    _distMatrix->at((_rowCount - 1) - i).at(_colCount - 1) = i * _inDel;
  }

  // Fill bottom row right to left w/ multiples of inDel
  for (size_t j = 0; j < _colCount; j++) {
    _distMatrix->at(_rowCount - 1).at((_colCount - 1) - j) = j * _inDel;
  }

  for (size_t i = _rowCount - 1; i > 0; i--) {
    for (size_t j = _colCount - 1; j > 0; j--) {
      // Take value of diagonal down right + penalty
      match = _distMatrix->at(i).at(j)
            + penalty(_stringX.at(i - 1), _stringY.at(j-1));

      // Take value from below + indel
      remove = _distMatrix->at(i - 1).at(j) + _inDel;

      // Take value from right + indel
      insert = _distMatrix->at(i).at(j - 1) + _inDel;

      size_t optChoice = min(match, remove, insert);
      _distMatrix->at(i-1).at(j-1) = optChoice;
    }
  }
  return _distMatrix->at(0).at(0);
}

// Retrace distance matrix and align strings using optimal choices.
std::string EDistance::alignment() {
  std::string alignA = "";
  std::string alignB = "";
  std::string result = "";

  size_t i = 0;
  size_t j = 0;

  while ( i < (_rowCount - 1) || j < (_colCount - 1) ) {
    if ( i < (_rowCount - 1) && j < (_colCount - 1)
        && _distMatrix->at(i).at(j) == _distMatrix->at(i + 1).at(j + 1)
        + penalty(_stringX.at(i), _stringY.at(j)) ) {
      // Move down right = match/mismatch
      alignA += _stringX.at(i);
      alignB += _stringY.at(j);
      i++; j++;
    } else if ( i < (_rowCount - 1)
      && _distMatrix->at(i).at(j) == _distMatrix->at(i+1).at(j) + _inDel
          ) {
      // Move down = add gap to string y.
      alignA += _stringX.at(i);
      alignB += "-";
      i++;
    } else {
      // Move right = add gap to string x.
      alignA += "-";
      alignB += _stringY.at(j);
      j++;
    }
```

```
  }
  // Create and format final result string.
  for (size_t i = 0; i < alignA.length(); i++) {
    if (alignA.at(i) == '-' || alignB.at(i) == '-') {
      // Gap char found. Print chars and 2.
      result += alignA[i];
      result += " ";
      result += alignB[i];
      result += " 2\n";
    } else if (alignA[i] != alignB[i]) {
      // Mismatch found. Print chars and 1.
      result += alignA[i];
      result += " ";
      result += alignB[i];
      result += " 1\n";
    } else {
      // Match found. Print chars and 0.
      result += alignA[i];
      result += " ";
      result += alignB[i];
      result += " 0\n";
    }
  }

  return result;
}
```

7 PS6: Random Writer

7.1 Objective

The objective of this assignment was to implement the class RandWriter that will act a text generator based off a probabilistic model by using Markov chains. The RandWriter class will take in a segment of input text called the *corpus* and it will generate a segment of text of a specified length that is very similar to the source material. For our assignment we were provided a copy of *The Adventures of Tom Sawyer* for use as source material.

7.2 Outcome

My outcome was a positive one for this assignment as my text generator produced text that was both similar in content to the source text and syntactially correct while still being quite different from the original source.

Seen below is an output from my text generator created by using the first few paragraphs of Chapter 1 of *The Adventures of Tom Sawyer*.

```
The old lady pulled her spectacles down and looked out among the tomato vi
nes and "jimpson" weeds that constituted the garden. No Tom. So she lifted
 up her voice at an angle calculated for distance and shouted:
"Y-o-u-u TOM!"
```

Figure 7.1: Output from the RandWriter class.

7.3 Implementation

My implementation of the RandWriter class was accomplished by first reading and dissecting the input text fed to the class constructor at run time. The input corpus is first redirected to stdin from the command line and then fed to the class constructor along with a the order of the Markov model represented by the parameter k.

After reading in the source corpus, the text was broken down into "k-grams" of size k by parsing k characters at a time from the source material and storing them as keys into the map **_symbolTable** . The **_symbolTable** map was comprised of a pair of two strings. The first string would represent the k-gram that was parsed and the second string would act as a container for the single characters that were seen immediately after each k-gram. If a duplicate k-gram was parsed from the text, the character that immediately followed would still be stored in the second string paired to that k-gram.

Once this process was completed, **_symbolTable** will be populated with all the k-grams of the order k, paired with a string that contains all of the characters seen immediately after each k-gram. In the event that an order k of 0 is specified for the Markov model, the input text is parsed character by character and the frequency of each character is stored on a separate map called **_charTable**.

After dissection of the corpus, new text is generated using the member functions of the RandWriter class. The generate() function takes two parameters, the first being the specified k-gram that the generated text should start with, and a the second being the length L which will dictate the length of the generated text.

Then generate() will proceed to produce an output string by emplacing the first k-gram directly to the output and then calling the function kRand() which will randomly select a character that was seen after that specific k-gram in **_symbolTable**. After finding a new random character, it is pushed to the output string and then a new k-gram is generated by dropping the first character of our initial k-gram and appending on our randomly generated character returned by kRand(). The lookup process then repeats using the new k-gram.

At every step, we emplace one new character at a time to the output string. The character chosen is predicated on what characters are seen after each k-gram in **_symbolTable**. This process continues until we have reached a string length specified by the parameter L. In the event that a RandWriter has created a model of order 0, the generate() function will simply push on randomly chosen characters to the output string whose frequency of selection are dictated by the map **_charTable**.

The other member functions of my RandWriter class provide specific information on the internal state of the Markov model that RandWriter has created. For example, the function orderK() will return the order of the model, freq() will return the number of instances a specific k-gram or following character has been seen in the corpus text, and printAlphabet() will print the breadth of characters seen in the corpus.

The insertion operator has also been overloaded and will print the contents of **_symbolTable** or **_charTable** respectfully depending on the order of the Markov model.

7.4 Lessons Learned

The main concepts that I took from this assignment was the idea of a Markov Model or Markov Chain. A Markov Model is a stochastic model that describes a sequence of states or events wherein the probability of a particular event occurring is dependent upon the previous state or event. In our assignment we emulated this process in a similar fashion by the use of k-grams and a symbol table. Our k-gram represents the current state that we are in where the symbol table contains the the probabilities of traversal to our next state.

The use of Markov Chains is very common in algorithms that deal with modeled prediction based on previously recorded results. These kinds of algorithms are commonly seen in programs that generate predictive text or use speech recognition. This Markov model also has many uses outside of programming as well, such as in the finance, chemistry, and physics.

7.5 Code Listing

7.5.1 Makefile

```
CC = g++ -g
CFLAGS = -Wall -Werror -pedantic --std=c++14
DEPS = RandWriter.hpp
LIBS = -lboost_unit_test_framework
SOURCES = TextWriter.cpp RandWriter.cpp test.cpp
LINT = /usr/local/bin/cpplint.py
LINTFLAGS = --filter=-runtime/references,-build/header_guard,-build/c++11 --extensions=cpp,hpp

all: TextWriter

TextWriter: TextWriter.o RandWriter.o
	$(CC) $(CFLAGS) -o $@ $^ $(LIBS)

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

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

test: test.o RandWriter.o
	$(CC) $(CFLAGS) -o $@ $^ $(LIBS)

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

lint: $(SOURCES) $(DEPS)
	$(LINT) $(LINTFLAGS) $(SOURCES) $(DEPS)

clean:
	rm *.o TextWriter test
```

7.5.2 Driver Code

TextWriter.cpp

```
// Copyright 2022 - Ryan Casey
/**
* PS6 - Random Writer
* Instructor: James Daly
* Todays Date: 4/11/2022
* Due Date : 4/11/2022
* Program Description:
*
In this program I have emulated the Markov chain model which
can be used to created random predictive text based on an input corpus.

My implementation works by iterating over the source text char by char
and dissects the text into  all possible kgram strings that are the size of
```

```
the model order specified by the user.

Once all kgrams have been dissected from the input text, all chars that
follow each kgram are then tracked and stored into the symbol table of my
class. After instantion of the RandWriter class, its member symbol table
will be fully populated with all possible kgrams and the characters that
immediately follow each kgram whereever it appears in the source text.

The member methods of the RandWriter class include informational methods
that can inform the user of the number of times a kgram appears and the number
of times a specific character follows a kgram. Lastly, the RandWriter
generate() function will produce a sample text of randomly generated
words via the prediction model created via the markov chain. The order
of the markov model, the length of the outputted sample text, and a source
text file to extract the corpus from, are all passed by the user via
the command prompt.
*
* Created by: Ryan Casey
**/
#include <iostream>
#include "RandWriter.hpp"

int main(int argc, char* argv[]) {
  // Get cmd line args.
  int k, l;
  k = std::stoi(argv[1]);
  l = std::stoi(argv[2]);

  std::string inputString;
  std::string firstKChar = "";
  char ch;

  // Helper lambda func.
  auto pushChar = [] (char ch, std::string& inputString) {
    inputString.push_back(ch);
  };

  // Push chars into inputString until we hit EOF.
  while (std::cin.peek() != EOF) {
    std::cin.get(ch);
    pushChar(ch, inputString);
  }

  // Set first k chars for first kgram.
  for (int i = 0; i < k; i++) {
    firstKChar += inputString.at(i);
  }

  // Create model and generate text.
  RandWriter textWriter(inputString, k);
  std::cout << textWriter.generate(firstKChar, l) << std::endl;

  return 0;
}
```

7.5.3 Interface Files (.hpp)

RandWriter.hpp

```cpp
// Copyright 2022 - Ryan Casey
#pragma once
#include <string>
#include <iostream>
#include <map>
#include <regex>

class RandWriter {
 public:
  RandWriter(std::string text, int k);
  int orderK() const { return _order;}
  int freq(std::string kgram) const;
  int freq(std::string kgram, char c) const;
  char kRand(std::string kgram);
  std::string generate(std::string kgram, int L);

  friend std::ostream& operator<<(std::ostream& os, RandWriter& kgram);

  void printAlphabet() { std::cout << _alphabet << std::endl; }

 private:
  std::string _alphabet = "";
  std::map<char, int> _charTable;
  std::map<std::string, std::string> _symbolTable;
  int _order;
};
```

7.5.4 Implementation Files (.cpp)

RandWriter.cpp

```cpp
// Copyright 2022 - Ryan Casey
#include <iostream>
#include <exception>
#include <random>
#include <chrono>
#include <utility>
#include "RandWriter.hpp"
#define NUM_ASCII_CHAR 127

// Create a Markov model of order k from given text.
// Param 'text' is assumed to be of length 'k'.
RandWriter::RandWriter(std::string text, int k) {
  // Throw if k > text.length().
  if ( text.length() < static_cast<size_t>(k) ) {
    throw std::runtime_error(
    "Exception RandWriter(): string length must be at least size of order.\n");
  }

  // Set member fields.
```

```cpp
  _alphabet = text;
  _order = k;

  if ( k > 0 ) {
    // Traverse input text char by char.
    for (size_t i = 0; i < text.length(); i++) {
      // Add first char to gram.
      std::string gram;
      gram += text[i];

      // Add the k chars that follow the first char to gram.
      size_t j;
      for (j = 1; j < (static_cast<size_t>(k)); j++) {
        gram += text[(i + j) % text.length()];
      }

      // Add gram to _symbolTable. Increment its count if it already exists.
      std::pair<std::map
      <std::string, std::string>::iterator, bool> inserted;

      inserted =
      _symbolTable.insert(std::pair<std::string, std::string>(gram, ""));

      // Grab character that follows gram and save to symbol table.
      _symbolTable.at(gram) += text[(i + j) % text.length()];
    }
  } else {
    // If order = 0, populate char table with appearance/frequency of each char.
    for (auto ch : text) {
      std::pair<std::map<char, int>::iterator, bool> inserted;
      inserted = _charTable.insert(std::pair<char, int>(ch, 1));
      if (!inserted.second) {
        _charTable.at(ch) += 1;
      }
    }
  }
}

// Return number of occurrences of kgram in text.
int RandWriter::freq(std::string kgram) const {
  // throw an exception if kgram is not of length k.
  if (kgram.length() != static_cast<size_t>(_order)) {
    throw std::runtime_error(
    "Exception freq(string): Length of kgram must equal order of model.\n");
  }
  int occcurences = 0;

  // If order 0, return length of original input string.
  if (kgram.length() == 0) {
    return _alphabet.length();
  } else {
    // Otherwise return occurences of kgram.
    occcurences = _symbolTable.at(kgram).length();
  }
  return occcurences;
```

```cpp
}

// Return number of times that character c follows kgram.
// If order = 0, return num of times char c appears.
int RandWriter::freq(std::string kgram, char c) const {
  // throw an exception if length of kgram != order.
  if (kgram.length() != static_cast<size_t>(_order)) {
    throw std::runtime_error(
    "Exception freq(string,char): Size of kgram != order of model.\n");
  }
  int occurences = 0;

  // If order 0, return occurences of c in original input string.
  if (kgram.length() == 0) {
    for (auto ch : _alphabet) {
      if (ch == c) {
        occurences++;
      }
    }
  } else {
    // Otherwise return occurences of c following kgram.
    for (auto ch : _symbolTable.at(kgram)) {
      if (ch == c) {
        occurences++;
      }
    }
  }
  return occurences;
}

// Return random character following given kgram
char RandWriter::kRand(std::string kgram) {
  int seed = std::chrono::system_clock::now().time_since_epoch().count();
  std::minstd_rand randVal(seed);
  char randChar;

  // Throw an exception if kgram length != order of model.
  if (static_cast<int>(kgram.length()) != _order) {
    throw std::runtime_error(
    "Exception kRand(string): kgram size != order of model.\n");
  }

  if (_order != 0) {
    // Throw an exception if no such kgram.
    std::map<std::string, std::string>::iterator it;
    it = _symbolTable.find(kgram);
    if (it == _symbolTable.end()) {
      throw std::runtime_error(
      "Exception kRand(string): Could not locate kgram in symbol table.\n");
    }

    // Get a rand value that is within the num of chars that follow kgram.
    int rand = randVal() % static_cast<int>(_symbolTable.at(kgram).length());

    // Choose a random char from chars that follow kgram.
```

```
    randChar = _symbolTable.at(kgram).at(rand);
  } else {
    // Get a rand value that is within the num of chars in the input string.
    int rand = (randVal() % static_cast<int>(_alphabet.length()));

    // Choose a random char from chars that follow kgram.
    randChar = _alphabet.at(rand);
  }
  return randChar;
}
// Generate and return a string of length L characters
// by simulating a trajectory through the corresponding
// Markov chain. The first k characters of the newly
// generated string should be the argument kgram.
// Assume that L is at least k
std::string RandWriter::generate(std::string kgram, int L) {
  // Throw an exception if kgram is not equal to order of model.
  if (static_cast<int>(kgram.length()) != _order) {
    throw std::runtime_error(
    "Exception generate(string, int): kgram size != order of model.\n");
  }
  std::string currentGram, nextGram;
  std::string outputString = "";
  char nextChar;
  currentGram = kgram;

  // Add current gram to output string.
  outputString += currentGram;

  // While we still have room in output string.
  while (static_cast<int>(outputString.length()) < L) {
    // If order = 0, populate output string with rand chars from input string.
    if (_order == 0) {
      for (int i = 0; i < L; i++) {
        outputString += kRand("");
      }
    } else {
      // Otherwise look up next char after current
      // gram and add to output string.
      nextChar = kRand(currentGram);
      outputString += nextChar;

      // Update nextGram by sliding over 1 char and adding nextChar.
      nextGram = "";
      for (size_t i = 1; i < currentGram.length(); i++) {
        nextGram += currentGram.at(i);
      }
      // Update currentGram with nextGram.
      nextGram += nextChar;
      currentGram = nextGram;
    }
  }
  return outputString;
}

// Overload the stream insertion operator and display
```

```
// the internal state of the Markov Model. Print out
// the order, the alphabet, and the frequencies of
// the k-grams and k+1-grams.
std::ostream& operator<<(std::ostream& os, RandWriter& kgram) {
  kgram.printAlphabet();
  std::cout << "----- k-grams of order: "
  << kgram._order << " -----" << std::endl;

  if (kgram._order == 0) {
    std::cout << "Chars in string:\n";
    for (auto ch : kgram._charTable) {
      std::cout << ch.first << ": Frequency: " << ch.second << std::endl;
    }
  }
  for (auto gram : kgram._symbolTable) {
    std::cout << gram.first << ": Chars that follow: [";
    for (size_t i = 0; i < gram.second.length(); i++) {
      std::cout << gram.second[i];
      if (i < gram.second.length() - 1) {
        std::cout << ", ";
      }
    }
    std::cout << "]" << std::endl;
  }
  return os;
}
```

8 PS7: Kronos Time Parsing

8.1 Objective

The objective of this assignment was to parse a service log from the Kronos InTouch device and to generate an output report that details successful boot operations and the duration of the boot cycle itself. For this assignment we were also tasked with utilizing the Boost regex library and the Boost date/time library to accurately parse the date and time stamps included in the service log.

8.2 Outcome

I was able to successfully complete all required tasks for this assignment and was able to verify proper functionality and accuracy by parsing a collection of five different Kronos InTouch service logs. Seen below is the output report created from parsing a Kronos InTouch device log. As you can see, the report details the date and time of the start of the boot sequence, the completion of the sequence, and how long that difference in time was. If a boot sequence was not completed successfully, the report also details an incomplete boot.

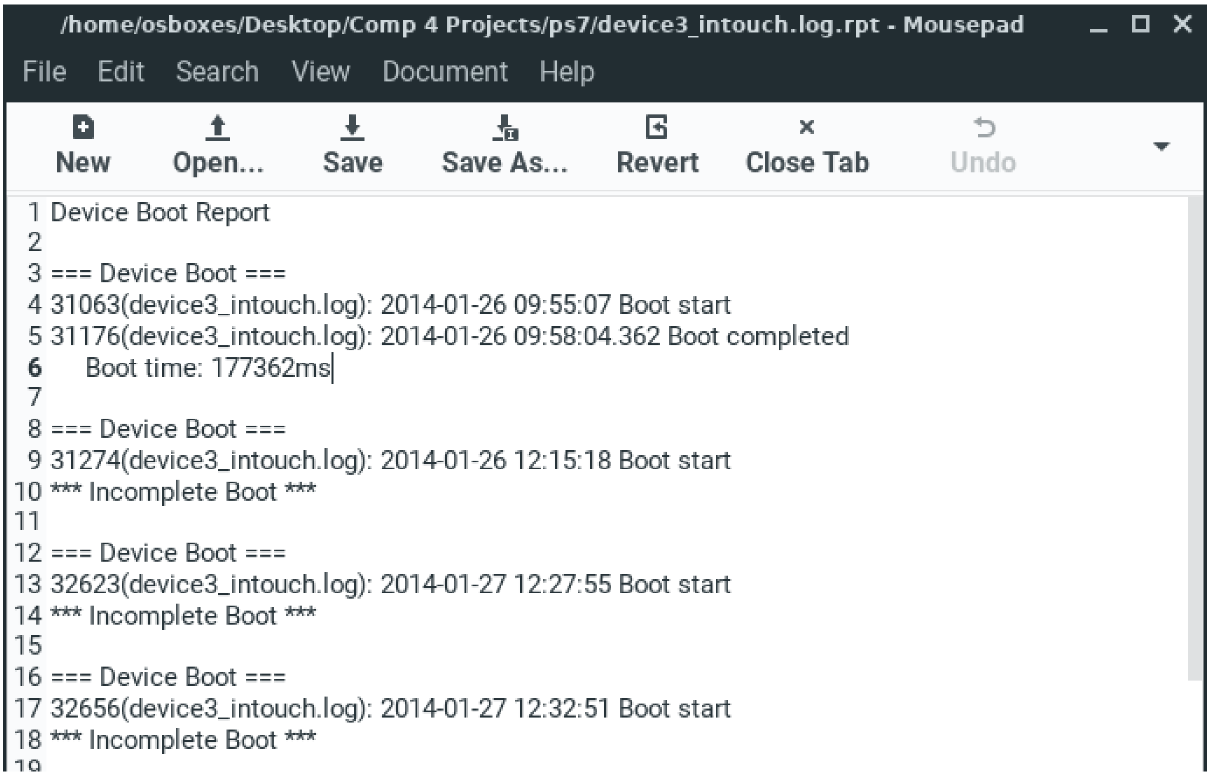

Figure 8.1: Output from the RandWriter class.

8.3 Implementation

The implementation of my log parsing program was accomplished with the use of a three strings that were then used in the creation of two specific regular expressions, one that filtered for the boot start event and one to filter for the boot completion event.

The first string created was specifically created to target the time stamp that preceded every message in the source log file. This string called **dateTimeRegex** looks for four digits, a space or dash, followed by two digits, another space or dash, then a final two digits. This allowed for the identification of set of characters in year, month, day format (Such as: *YYYY-MM-DD* or *YYYY MM DD*).

The second and third strings were targeted to the specific messages produced for a boot event or completion event. The start message that was filtered for was "*(log.c.166) server started*", and this was assigned to the string **startMsgRegex**. The completion message that was filtered for was "*oejs.AbstractConnector:Started SelectChannelConnector*", with this being assigned to **successMsg**. (Note: The success string was broken apart into an A and B component for better readability and linting purposes.)

Using these strings, two larger strings were created that were comprised of the date and time string and the start or completion message. Both of these larger combined strings were then used for the creation of two regular expressions called **reStart** and **reSuccess**. Once these regex were created, my program began to parse line by line, the Kronos InTouch device log that was fed to the program via input redirection from the command line.

When a boot start message is correctly identified using the regex **reStart**, a flag **startFound** is set. As the program continues, it will then look for a completion message that matched the regex **reSuccess**. If a success message is found after a start message, the difference in time between both messages is calculated using boost::posix_time from the boost date and time library. When another boot start message is encountered prior to a success message, this indicates that the boot did not complete successfully.

As we move line by line and determine whether a successful or incomplete boot had occurred, the number of lines read by the program and the results of each event are outputted to an output stream called **logFile** along with the date and time and time difference for a successful boot cycle. The output stream **logFile** is directed to output to our final report file.

Once the end of file has been reached, one final check is made to see if our last regex match was from a boot start message. If so this indicated that the final boot was incomplete.

8.4 Lessons Learned

The main concept I took away from this assignment was the use of regular expressions. Prior to this assignment my normal methodology for the comparison of two strings was a direct comparison using the == operator. This however is not the most robust method of comparison and does not take into consideration small differences in formatting nor does it encompass situations where parts of the string change while other parts are consistent, such as a a date and time stamp.

The use of a regular expressions in this assignment allowed for a broader and more accurate parsing of the log file and also allowed for the extraction of the time stamp for each message which was critical for the calculation of the boot sequence time.

Regular expressions like the ones used here are very useful for the parsing of large batches of slightly variable information and are frequently used in input sanitation to ensure user input has met certain requirements to be deemed acceptable.

8.5 Code Listing

8.5.1 Makefile

```
CC = g++
CFLAGS = -Wall -Werror -pedantic --std=c++14
DEPS =
LIBS = -lboost_unit_test_framework -lboost_regex -lboost_date_time
SOURCES = main.cpp
LINT = /usr/local/bin/cpplint.py
LINTFLAGS = --filter=-runtime/references,-build/header_guard,-build/c++11 --extensions=cpp,hpp

all: ps7

ps7: main.o
	$(CC) $(CFLAGS) -o $@ $^ $(LIBS)

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

lint: $(SOURCES) $(DEPS)
	$(LINT) $(LINTFLAGS) $(SOURCES) $(DEPS)

clean:
	rm *.o ps7
```

8.5.2 Driver Code

main.cpp

```
// Copyright 2022 - Ryan Casey
/**
* PS7 - Kronos Time Clock
* Instructor: James Daly
* Todays Date: 4/20/2022
* Due Date : 4/20/2022
* Program Description:

In this program I have implemented a report generator
that parses a log file and outputs to a report file.
The log file is passed via command line arguments and this
progam will parse the data line by line looking for instances
of a predefined boot and boot success std::strings by means of
comparison to a regular expression.

If a boot event is found and matched to a success event in the log,
```

```
the time stamps of both events are used to determine the duration of
the boot sequence. If no matching success event is found for a boot
event, an incomplete boot is outputted.

Parsing w/ regex was accomplished via the use of the boost
regex library, while date/time manipulation was accomplished
via the use of the boost date_time library.

*
* Created by: Ryan Casey
**/
#include <iostream>
#include <fstream>
#include <string>
#include <boost/regex.hpp>
#include "boost/date_time/gregorian/gregorian.hpp"
#include "boost/date_time/posix_time/posix_time.hpp"

int main(int argc, char* argv[]) {
  // Open log file and insert into std::ifstream.
  std::ifstream ifs;
  ifs.open(argv[1], std::ifstream::in);

  // Create report file using passed log file name.
  std::string fileName = argv[1];
  fileName += ".rpt";
  std::ofstream logFile(fileName);

  // Regex filter strings for date, time, boot start, and boot success.
  std::string dateTimeRegex =
  "(\\d{4}[- ]\\d{1,2}[- ]\\d{1,2} \\d{2}:\\d{2}:(?:\\d{2}|\\d{2}\\.\\d{3}))";
  std::string startMsgRegex =
  ": \\(log\\.c\\.166\\) server started ";
  std::string successMsgA =
  ":INFO:oejs\\.AbstractConnector:Started";
  std::string successMsgB =
  " SelectChannelConnector@0\\.0\\.0\\.0:9080";

  // Combine date, time, and messages into single strings for regex.
  std::string startRegex, successRegex;
  startRegex += dateTimeRegex + startMsgRegex;
  successRegex += dateTimeRegex + successMsgA + successMsgB;

  // Create regex's using combined strings.
  boost::regex reStart, reSuccess;
  try {
    reStart = boost::regex(startRegex);
    reSuccess = boost::regex(successRegex);
  } catch(boost::regex_error& exc) {
    std::cerr << "Regex constructor failed with code "
    << exc.code() << std::endl;
    exit(1);
  }

  // Init vars for parsing.
  std::string currentLine;
  size_t linesRead = 0;
```

```
bool startFound = false;
bool successFound = false;
boost::smatch startMatch;
boost::smatch successMatch;
boost::posix_time::ptime startTime;
boost::posix_time::ptime successTime;
boost::posix_time::time_duration td;

// Parse log file and output boot successes/time/failures.
logFile << "Device Boot Report\n\n";

while (!ifs.eof()) {
  // Read line by line.
  getline(ifs, currentLine);
  linesRead++;

  // Look for a boot message if we havent seen one yet.
  // Output log line number, file name, and time stamp of boot.
  if (!startFound && (regex_match(currentLine, reStart))) {
    regex_match(currentLine, startMatch, reStart);
    startTime = boost::posix_time::time_from_string(startMatch[1]);
    logFile << "=== Device Boot ===\n";
    logFile << linesRead << "(" << argv[1] << "): " << startMatch[1]
    << " Boot start" << std::endl;

    startFound = true;

    // If we see another boot message before a success, boot failed.
  } else if (startFound && (regex_match(currentLine, reStart))) {
    regex_match(currentLine, startMatch, reStart);
    logFile << "*** Incomplete Boot ***\n\n";
    startFound = false;
  }
  // Look for a success message if we've found a boot message.
  // Output log line number, file name, and time stamp of boot success.
  if (startFound && (regex_match(currentLine, reSuccess))) {
    regex_match(currentLine, successMatch, reSuccess);
    successTime = boost::posix_time::time_from_string(successMatch[1]);
    logFile << linesRead << "(" << argv[1] << "): "
    << successMatch[1] << " Boot completed" << std::endl;

    successFound = true;
  }

  // If we find a boot message paired with a success message,
  // calulate the boot time.
  if (startFound && successFound) {
    td = successTime - startTime;
    logFile << "     Boot time: " << td.total_milliseconds() << "ms\n\n";
    startFound = false;
    successFound = false;
  }
}
// If we reach eof and have found no matching success, boot failure.
if (startFound && !successFound)
  logFile << "*** Incomplete boot ***\n\n";
```

```
  // Close files.
  ifs.close();
  logFile.close();

  return 0;
}
```