COMP IV: C++ Project Portfolio

Brent Garey

COMPUTING.2040
UMASS LOWELL

Table of Contents:

PS1: Linear Feedback Shift Register

Part A- Implement a 16bit LFSR

Part B- PhotoMagic Transformation

Objective:

Encrypt a .png file using a 16bit LFSR as a seed. The output will change the pixels of the image so that it cannot be viewed / restored unless you put in the same LFSR that was used as a seed to encrypt it.

Key algorithms, data structures, or OO designs that were central to this assignment:

Utilizing XOR allowed us to be able to revert the image back to its original form.

Understanding SFML and how it stores images and pixels allowed me to save the RGB value of each pixel and XOR it with the LFSR.

Correctly implementing the LFSR was critical since errors would lead to the "password" not doing it's intended functionality.

What was learned:

- XOR is an operation that is its own inverse.
    - Applying XOR is easily undone by another XOR of the same value.
        - This inherently acts as a key.

- How to access the pixels of an image stored in SFML.
    - SFML stores a picture as a collection of pixels. Each pixel has three integers to represent the color values of RGB. Manipulating the RGB values renders the image unreadable.

FibLFSR.h

```
class FibLFSR {
       public: FibLFSR(string seed);
       FibLFSR();
       ~FibLFSR();
       int step();
       int generate(int k);
       string getFLFSR() const;
       friend ostream& operator<<(ostream&, const FibLFSR&);

       private: string flfsr;
};
```

FibLFSR.cpp

```
//constructor to create LFSR with
//the given initial seed
FibLFSR::FibLFSR(string seed) {
       flfsr = seed;
}

//empty constructor should nothing be given
FibLFSR::FibLFSR(){
       flfsr = '0';
}

//destructor in case we need to free memory (not for this problem)
FibLFSR::~FibLFSR() { cout << "FibLFSR destructor called" << endl; }

int FibLFSR::step() {

              char* s;
              s = &(flfsr.at(1));
              //store the first bit that will be shifted
              char first_bit = flfsr.at(0);

              //store each tap position
              char tap1 = flfsr.at(2);
              char tap2 = flfsr.at(3);
              char tap3 = flfsr.at(5);

              //convert them to int to compute xor
              int First_bit = ((int) first_bit) - 48;//atoi(&first_bit);
              int Tap1 = ((int) tap1) - 48;//atoi(&tap1);
              int Tap2 = ((int) tap2) - 48;//atoi(&tap2);
              int Tap3 = ((int) tap3) - 48;//atoi(&tap3);

              //compute the new bit
              int first_xor = First_bit ^ Tap1;
              int second_xor = first_xor ^ Tap2;
              int third_xor = second_xor ^ Tap3;

              //shift the bits
              flfsr = s;

              //convert from int to string
              string last_bit;
```

```
		if (third_xor == 1) {
			last_bit = "1";
			}
		else {
			last_bit = "0";
			}
		//append the last bit
		flfsr = flfsr + last_bit; //to_string(third_xor);

		return third_xor;
	}
	//simulate one step and return
	//the new bit as 0 or 1

	int FibLFSR::generate(int k) {
		int value = 0;
		for (;k > 0; k--)
		{
			value = (value * 2) + step();
			}
		return value;
		}
	//simulate k steps and return k-bit integer

	string FibLFSR::getFLFSR() const { return flfsr; }
	//accessor function returns the register value
	//overload << in order to display current register value
	ostream& operator<< (ostream& out, const FibLFSR& f) {
	out << f.getFLFSR();
	return out;
}
```

PhotoMagic.cpp

```
#include <SFML/System.hpp>
#include <SFML/Window.hpp>
#include <SFML/Graphics.hpp>
#include "FibLFSR.cpp"

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

int main(int argc, char* argv[])
{
 //lets open the image and store it
 sf::Image image;
 if(!image.loadFromFile("input-file.png"))
	return -1;
 FibLFSR password(argv[3]);

 transform(image, &password);

// sf::Color p;

sf::Vector2u size = image.getSize();
sf::RenderWindow window(sf::VideoMode(size.x, size.y), "Encryption / Decryption");
// for(int x = 0; x < size.x; x++){
//     for(int y = 0; y < size.y; y++){
//             p = image.getPixel(x, y);
//             p.r = 255 - p.r;
//             p.g = 255 - p.g;
//             p.b = 255 - p.b;
//             image.setPixel(x,y,p);
//             }
//     }

 sf::Texture texture;
 texture.loadFromImage(image);

 sf::Sprite sprite;
 sprite.setTexture(texture);

 while (window.isOpen())
{
       sf::Event event;
       while(window.pollEvent(event))
       {
              if(event.type == sf::Event::Closed)
                     window.close();
       }
       window.clear(sf::Color::White);
       window.draw(sprite);
       window.display();
}

 if(!image.saveToFile("input-file.png"))
       return -1;
 if(!image.saveToFile("output-file.png"))
       return -1;

 return 0;
}
void transform(sf::Image& image, FibLFSR* password)
{

       sf::Color p;
       sf::Vector2u size = image.getSize();
       for(int x = 0; x < size.x; x++){
              for(int y = 0; y < size.y; y++){
                     p = image.getPixel(x,y);
                     p.r =  (p.r) ^ (password->generate(8)); //255 - p.r;
                     p.g =  (p.g) ^ (password->generate(8)); //255 - p.g;
                     p.b =  (p.b) ^ (password->generate(8)); //255 - p.b;
                     image.setPixel(x,y,p);
                     }
              }

}
```

PS2: N-Body Simulation

Part A- CelestialBody.cpp & Universe.cpp

Part B- NBody.cpp

Objective:

- Implement a physics simulation of planets in our solar system by:
    - Create a CelestialBody class that will hold data for each planet.
    - Create a Universe class that is full of every CelestialBody created.
        - The Universe class will then manipulate the xPosition and yPosition of each CelestialBody based on the other CelestialBodys.

Key algorithms, data structures, or OO designs that were central to this assignment:

- Classes
    - Each planet was stored as a CelestialBody class.
- Vectors
    - The Universe class stored CelestialBodys via vector<CelestialBody*>
        - Holding a vector of pointers to each CelestialBody allowed the universe to constantly access and manipulate each CelestialBody.

This assignment solidified my familiarity with object orientated programming by creating two classes and constantly having to check and update their date members.

CelestialBody.cpp

```
//implement a celestialbody class
class CelestialBody{
	public:
	//value constructor
	CelestialBody(double xPos, double yPos, double xVelo,
	double yVelo, double Mass, string fileName, double universeradius, sf::Vector2u
	Size){
	xPosition = xPos;
	yPosition = yPos;
	xVelocity = xVelo;
	yVelocity = yVelo;
	mass = Mass;
	filename = fileName;
	universeRadius = universeradius;
	size = Size;
	//load image, texture and sprite
	image.loadFromFile(filename);
	texture.loadFromImage(image);
	sprite.setTexture(texture);
	}

	//setter for netForce
	void set_netForce(double force){
	netForce = force;}

	//empty constructor
	CelestialBody(void){
	xPosition = 0;
	yPosition = 0;
	xVelocity = 0;
	yVelocity = 0;
	mass = 0;
	filename = "empty";
	}
	//getter functions to return private members
	double get_xPosition(void){
	return xPosition;}

	double get_yPosition(void){
	return yPosition;}

	double get_xVelocity(void){
	return xVelocity;}

	double get_yVelocity(void){
	return yVelocity;}

	double get_mass(void){
	return mass;}

	double get_netForce(void){
	return netForce;}

	double get_xAcceleration(void){
	return xAcceleration;}
```

```
double get_yAcceleration(void){
return yAcceleration;}

void setPosition(void){
xPosition = ((xPosition / universeRadius) * (size.x / 2)) + (size.x / 2);
yPosition = ((yPosition / universeRadius) * (size.y / 2)) + (size.y / 2);
sprite.setPosition(xPosition, yPosition);
}

//mutator function to modify velocity
void set_xPosition(double position){
xPosition = position;}
void set_yPosition(double position){
yPosition = position;}
void set_xVelocity(double velocity){
yVelocity = velocity;
}
void set_yVelocity(double velocity){
xVelocity = velocity;
}
void set_xAcceleration(double accel){
xAcceleration = accel;}
void set_yAcceleration(double accel){
yAcceleration = accel;}
//calculate the net force (Fx and Fy) at current time t acting on
//this celestial body object
//void set_netForce(){

//moves object due to velocity for given time
void step(double seconds);
//overload insertion operator to read in values
friend istream& operator>>(istream& in, CelestialBody& celestialBody){
in >> celestialBody.xPosition >> celestialBody.yPosition;
in >> celestialBody.xVelocity >> celestialBody.yVelocity;
in >> celestialBody.mass;
in >> celestialBody.filename;

if(!celestialBody.image.loadFromFile(celestialBody.filename))
       return in;
celestialBody.texture.loadFromImage(celestialBody.image);
celestialBody.sprite.setTexture(celestialBody.texture);
celestialBody.sprite.setPosition(celestialBody.xPosition,celestialBody.yPosition);

return in;
}

friend ostream& operator<<(ostream& out, CelestialBody& celestialBody){
out << "Xposition: " << celestialBody.xPosition << endl;
out << "Yposition: " << celestialBody.yPosition << endl;
out << "xvelocity: " << celestialBody.xVelocity << endl;
out << "yvelocity: " << celestialBody.yVelocity << endl;
out << "mass: " << celestialBody.mass << endl;
return out;
}

virtual void draw(sf::RenderWindow& window){
 window.draw(sprite);
}
```

```
        private:
        double xPosition;
        double yPosition;
        double xVelocity;
        double yVelocity;
        double mass;
        string filename;
        sf::Image image;
        sf::Texture texture;
        sf::Sprite sprite;
        double universeRadius;
        sf::Vector2u size;
        double netForce;
        double xAcceleration;
        double yAcceleration;
};
```

Universe.cpp

```
class Universe
{
       public:
       //default constructor
       Universe(sf::RenderWindow& window)
       {
        cin >> numPlanets;
        cin >> radius;
        //for each planet..
        for(; numPlanets > 0; numPlanets--)
        {
         cin >> xpos;
         cin >> ypos;
         cin >> xvelo;
         cin >> yvelo;
         cin >> mass;
         cin >> filename;
         size = window.getSize();

         //create a new planet with the info got
         CelestialBody* planet = new CelestialBody(xpos,ypos,xvelo,yvelo,mass,filename,
       radius, size);
         planets.push_back(planet);
        }
       }

       virtual void draw(sf::RenderWindow& window)
       {
        for(vector<CelestialBody*>::iterator it = planets.begin(); it!=
       planets.end();it++)
         {
              (*it)->draw(window);
         }
       }

       void step(double seconds){

       double deltaX;
       double deltaY;
       double deltaR;
       double force;
       double forceY;
       double forceX;
       double netForce;
       double xAccel;
       double yAccel;
       double xVeloNew;
       double yVeloNew;
       double xPosNew;
       double yPosNew;

        for(int i = 0; planets[i+1] != NULL; i++)
       {
       //get the difference in positions from i to the next one
       deltaX = planets[i]->get_xPosition() - planets[i+1]->get_xPosition();
       deltaY = planets[i]->get_yPosition() - planets[i+1]->get_yPosition();
deltaR = sqrt((deltaX * deltaX)+(deltaY * deltaY));

//get and set netforce
force = (Grav_const * (planets[i]->get_mass()) * (planets[i+1]->get_mass())) /
(deltaR * deltaR);
forceY = (force * deltaY) / (deltaR);
forceX = (force * deltaX) / (deltaR);
netForce = forceX + forceY;
planets[i]->set_netForce(netForce);
planets[i+1]->set_netForce(netForce);

//get and set acceleration
xAccel = forceX / planets[i]->get_mass();
yAccel = forceY / planets[i]->get_mass();
planets[i]->set_xAcceleration(xAccel);
planets[i]->set_yAcceleration(yAccel);

//calculate new velocity using the acceleration and time
xVeloNew = planets[i]->get_xVelocity() + (seconds * xAccel);
yVeloNew = planets[i]->get_yVelocity() + (seconds * yAccel);
planets[i]->set_xVelocity(xVeloNew);
planets[i]->set_yVelocity(yVeloNew);

//calculate new position
xPosNew = planets[i]->get_xPosition() + (seconds * planets[i]->get_xVelocity());
yPosNew = planets[i]->get_yPosition() + (seconds * planets[i]->get_yVelocity());
planets[i]->set_xPosition(xPosNew);
planets[i]->set_yPosition(yPosNew);
}

}

//friend istream& operator>>(istream& in, Universe universe){
//in >> universe.numPlanets >> universe.radius;
//in >> universe.xpos >> universe.ypos;
//in >> universe.xvelo >> universe.yvelo;
//in >> universe.mass >> universe.filename;
//return in;
//}

double get_G(void){
return Grav_const;}
friend ostream& operator<<(ostream& out, Universe universe){
 for(vector<CelestialBody*>::iterator it = universe.planets.begin(); it!=
universe.planets.end(); it++)
 {
       out << **it << endl;
 }
return out;
}
private:
int numPlanets;
double radius;
sf::Vector2u size;
//make a vector of celestialbodies
vector<CelestialBody*> planets;

//temp variables to hold scanned values
```

```
        double xpos,ypos,xvelo,yvelo,mass;
        string filename;

        //values for forces
        const double Grav_const = (6.67*pow(10,-11));
}
```

PS3: Synthesizing a Plucked String Sound

Part A- CircularBuffer.cpp

Part B- StringSound.cpp & KSGuitarSim.cpp

Objective: Create a program that will generate Guitar string sounds based off input from the keyboard. You will need a CircularBuffer data structure to implement KSGuitarSim.cpp that will be used to open the corresponding sounds, hold the frequencies, and play the sound to the user.

Key algorithms, data structures, or OO designs that were central to this assignment:

- Custom CircularBuffer class
  - This storage container needed to act as a ring, so that accessing data from one end will allow access to the other end (back+1 = front & front-1 = back)
  - Modulo was utilized to achieve this cyclic nature.

Part B involved utilizing memory to store the correct frequencies.

What was learned:

I learned that SFML is immensely helpful towards allowing us to load and store files into vectors. This was extremely helpful when it came to playing the "sound" of the corresponding frequency.

CircularBuffer.h:

```
class CircularBuffer{

	public:
	CircularBuffer(int capacity); //constructor given a max

	int size();		//return number of items currently in buffer
	bool isEmpty();		//does size = 0
	bool isFull(){return Size == Capacity;}	//does size=capacity
	void enqueue(int16_t x);//add item x to the end
	int16_t dequeue();	//delete and return item from the front
	int16_t peek();		//return (but do not delete) item from the front

	//friend ostream& operator<<(ostream& out, const CircularBuffer& item);
	//private:
	int Capacity;
	int Size;
	vector<int16_t> buffer;
	//int front;
	//int back;
};
```

CircularBuffer.cpp

```
CircularBuffer::CircularBuffer(int capacity) //constructor given a max
{
	if(capacity < 1)
	{
		throw std::invalid_argument("CircularBuffer constructor: capacity must be
greater than zero.");
	}else{
	buffer.reserve(capacity);
	Capacity = capacity;
	Size = 0;
	}
}

int CircularBuffer::size()//return number of items currently in buffer
{
	return Size;
}
bool CircularBuffer::isEmpty()//does size = 0
{
	if (Size == 0)
	{return true;}
	else{return false;}
}

void CircularBuffer::enqueue(int16_t x)//add item x to the end
{
	if(isFull()){
	throw std::runtime_error("enqueue: cant enqueue to full buffer");
```

```
        }
        buffer.push_back(x);
        Size++;



}

int16_t CircularBuffer::dequeue()//delete and return item from the front
{
        if(isEmpty())
        {
        throw std::runtime_error("dequeue: can't dequeue from empty ring");
        }
        int16_t item = buffer.front();
        int i = 0;
        for(vector<int16_t>::iterator it = buffer.begin(); it != buffer.end(); it++, i++)
        {
                buffer[i] = buffer[i+1];
        }
        Size--;
        return item;

}
int16_t CircularBuffer::peek()//return (but do not delete) item from the front
{
        if(isEmpty()){
        throw std::runtime_error("peek: can't peek from empty ring");
        }
        int16_t item = buffer.front();
        return item;

}
```

StringSound.cpp

```
#include "StringSound.h"
StringSound::StringSound(double frequency){      //create a guitar string sound of the
given
                                          //frequency using a sampling rate of 44,100
//make an exception to make sure its >0
//assign it to variable,
if(frequency < 0){
       throw std::invalid_argument("Frequency cannot be lower than 0");
}
int buff = ceil(SAMPLING_RATE/frequency);
buffer = new CircularBuffer(buff);
for(int i = 0; i < buff; i++)
{buffer->enqueue(0);}

count = 0;
}
StringSound::StringSound(vector<int16_t> init)  //create a guitar string with size
{                                  //and initial values given by vector
       if(init.empty())
       throw std::invalid_argument("vectore must be empty");

       buffer = new CircularBuffer(init.size());
       for(int i = 0; i < init.size();i++)
       {buffer->enqueue(init.at(i));}

count = 0;
}
void StringSound::pluck(void)            //replace the buffer with random values
{
       std::random_device rd;
while(!(buffer->isEmpty()))
       buffer->dequeue();

while(!(buffer->isFull())){
int16_t rng = -32768 + (rd() % ((32767 - (-32768)) + 1));
buffer->enqueue(rng);
}

}
void StringSound::tic(void){             //advance the simulation one time step
auto return_enqueue = [&](int16_t x) {buffer->enqueue(x); };

int16_t step = ((buffer->dequeue() + buffer->peek()) / 2) * DECAY_FACTOR;
count++;
return_enqueue(step);
}

int16_t StringSound::sample(void){ //return the current sample
return buffer->peek();
}
int StringSound::time(void){             //return number of times tic was called
return count;
}
```

KSGuitarSim.cpp

```
vector<int16_t> makeSamples(StringSound gs) {
	std::vector<int16_t> samples;

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

	return samples;
}
int main() {

  //modded
  auto getFreq = [](int i) {
	return (440 * pow(2, (i-24) / 12)); };


  //vector of samples
  vector<int16_t> samples;
  double freq;

//vector of 37 vector samples
  vector<vector<int16_t>> vec_Samples;
	//fill it up
	for(int i = 0; i < 37; i++){
	freq = getFreq(i);
	StringSound a = StringSound(freq);
	samples = makeSamples(a);
	vec_Samples.push_back(samples);
	}

	//vector of soundbuffers
  vector<sf::SoundBuffer> vec_SoundBuffers;
	//load each sample vector
	for(int j = 0; j < 37; j++){
	sf::SoundBuffer buf1;
	buf1.loadFromSamples(&vec_Samples.at(j)[0], vec_Samples.at(j).size(), 2,
SAMPLES_PER_SEC);
	vec_SoundBuffers.push_back(buf1);
	}

	//vector of sounds
  vector<sf::Sound> vec_sounds;
	//load each sound from the soundbuffer
	for(int k=0; k < 37; k++){
	sf::Sound sound1;
	sound1.setBuffer(vec_SoundBuffers[k]);
	vec_sounds.push_back(sound1);
	}

  sf::RenderWindow window(sf::VideoMode(300, 200), "SFML Plucked String Sound Lite");
  sf::Event event;
```

```
    std::string keyboard = "q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,./'";

    while (window.isOpen()) {
           while (window.pollEvent(event)) {
                  switch (event.type) {
                        case sf::Event::Closed:
                         window.close();
                         break;
                        case sf::Event::TextEntered:
                       {      std::string user_input;
                              user_input += event.text.unicode;
                              std::cout << user_input << " note" << std::endl;
                              int index = keyboard.find(user_input);
                              if(index < 0)
                                     break;
                              vec_sounds[index].play();
                       }

                       default:
                         break;
                  }
                  window.clear();
                  window.display();
           }
    }
    return 0;
  }
```

PS4: Markov Model of Natural Language

<u>Objective: </u>Implement a Markov Model structure that will keep track of

1. kGrams (length k grouping of consecutive characters)
2. The subsequent character

This will be used to calculate probability for each character whenever a kGram is found to give a random sample of probable predictions (of a given string).

Key algorithms, data structures, or OO designs that were central to this assignment:

- Map<string,int>
    - With the string set as a key, upon scanning the input there would be a check to see if there was a map already initialized for a kGram.
        - If there wasn't, set the int to 1 to indicate this is the first time seeing this grouping of characters.
        - Else, initialize a map for that kGram as 1 to indicate this is the first time seeing it.

```
if (kGramFreq.find(kGram) == kGramFreq.end()) {
// set the freq to 1(first time seeing)
kGramFreq[kGram] = 1;

else {  // if it gets this far.. its in the map
kGramFreq[kGram] = ++kGramFreq[kGram];
```

For both cases, I stored the subsequent characters as a separate map to hold the probability of each character that follows a kgram.

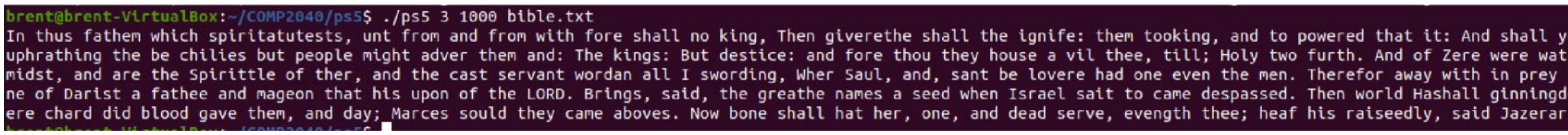
```
theoretical_char = word.substr(max + 1, 1);

kG[kGram].insert(std::make_pair(theoretical_char[0], 1));
```

What was learned:

Learning how to manipulate maps to store the kGrams and the alphabet (and their frequencies).

I specifically learned that the .find(kGram) will return .end() upon failing to find the map.

I then created a map to hold a <string, map <char, int> > to link the frequency of each subsequent character to each string. This implementation got me familiar with how to access and use the member functions associated with maps.



```
brent@brent-VirtualBox:~/COMP2040/ps5$ ./ps5 3 1000 bible.txt
In thus fathem which spiritatutests, unt from and from with fore shall no king, Then giverethe shall the ignife: them tooking, and to powered that it: And shall y
uphrathing the be chilies but people might adver them and: The kings: But destice: and fore thou they house a vil thee, till; Holy two furth. And of Zere were wat
midst, and are the Spirittle of ther, and the cast servant wordan all I swording, Wher Saul, and, sant be lovere had one even the men. Therefor away with in prey
ne of Darist a fathee and mageon that his upon of the LORD. Brings, said, the greathe names a seed when Israel sait to came despassed. Then world Hashall ginningd
ere chard did blood gave them, and day; Marces sould they came aboves. Now bone shall hat her, one, and dead serve, evength thee; heaf his raiseedly, said Jazerar
```

The program utilized kGrams of size 3 to generate an example of random output after reading the first 1000 characters of the bible to store probabilities.

```
class MModel {
	// creates a Markov model of order k for the specified text
public:
	MModel(string text, int k);

		// returns the order k of this Markov model
	int kOrder(void);

		// returns a string representation of the Markov model (as described below)
	//String toString();

		// returns the number of times the specified kgram appears in the text
	int freq(string kgram);

		// returns the number of times the character c follows the specified
		// kgram in the text
	 int freq(string kgram, char c);

		// returns a random character that follows the specified kgram in the text,
		// chosen with weight proportional to the number of times that character
		// follows the specified kgram in the text
	 char random(string kgram);

	 // generate 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.
	// Throw an exception if kgram is not of length k.
	// Assume that L is at least k.
	 string generate(string kgram, int L);

	 friend ostream& operator<< (ostream& out, MModel M) {
		out << "K is: " << M.K << endl;
		out << "L is: " << M.L << endl;
		//out << "kG is: " << M.kG << endl;
	 }

private:
	int K;
	int L;
	std::map<string, std::map<string, string>> kG;
};
```

```
MModel::MModel(std::string text, int k) {
  K = k;
  L = text.length();
  if (K > L || L == 0)
  throw std::invalid_argument("K cannot be greater than the length");
  word = text;

  // lamda expression to get our alphabet
  auto build_alphabet = [](std::string text, int L) {
  std::string Alphabet;
  for (int i = 0; i < L; i++) {
  if (Alphabet.find(text[i]) == std::string::npos)
  Alphabet += text[i];
  }
  return Alphabet;
  };
  alphabet = build_alphabet(text, L);
  std::vector <char> alpha;
  for (int i = 0; i < alphabet.length() ; ++i)
  alpha.push_back(alphabet[i]);

  std::string kGram;
  int max;
  std::vector<std::string> kGrams;
  std::string theoretical_char;
  // begin putting things into the vector! (this will our kgram)
  for (int i = 0; i < L; ++i, kGram = "") {
  bool counter = 0;
  if (i <= (L - K)) {  // as long as i+k is within bounds (doesnt loop)
  for (int j = 0; j < k; j++) {
  kGram += text[i + j];
  max = i + j;
  if (max == (word.length() - 1))
  max = 0;
  }
  } else {  // then this means that i+k LOOPS back!
  max = (i - (L - K) - 1);
  for (int j = i; j < L; j++)
  kGram += text[j];
  for (int p = 0; p <= max; p++)
  kGram += text[p];
  }
  theoretical_char = word.substr(max + 1, 1);
  kGrams.push_back(kGram);
  // lets use this theoretical character to count character
  std::map<char, int> alphaFreq;
  // if its not in the map,
  if (kGramFreq.find(kGram) == kGramFreq.end()) {
  // set the freq to 1(first time seeing)
  kGramFreq[kGram] = 1;
  // this is what came after the first kgram
  alphaFreq.insert(std::make_pair(theoretical_char[0], 1));
  // now lets create an index of the string, assign what we found to it.
  kG[kGram] = alphaFreq;
  } else {  // if it gets this far.. its in the map
  kGramFreq[kGram] = ++kGramFreq[kGram];
  // and adjust the according alphabet frequency
  // check if this theorietical character is already being counted
  if (kG[kGram].find(theoretical_char[0]) == kG[kGram].end()) {
  kG[kGram].insert(std::make_pair(theoretical_char[0], 1));
  } else {  //  just add 1 to the char freq
  alphaFreq.insert(std::make_pair(theoretical_char[0],
  ++(kG[kGram].at(theoretical_char[0]))));
  //  kG[kGram] = alphaFreq;
  }
  }
  //  we now have map['bbb'] = 1 or 2
  }
  int MModel::kOrder(void) {
  return K;
  }
  //  returns number of times the k-gram was found in the original text
  //  throw exception if kgram is not of length k
  int MModel::freq(std::string kgram) {
  if (kgram.length() != K)
  throw(std::invalid_argument("kgram is not of length k"));
  if (kGramFreq.find(kgram) != kGramFreq.end()) {
  return kGramFreq[kgram];
  } else {  //  else, the kgram isnt in the map
  return 0;
  }
  }
  //  returns the number of times the kgram was followed by
  // (throw an exception if kgram is not of length k)
  int MModel::freq(std::string kgram, char c) {
  if (kgram.length() != K)
  throw(std::invalid_argument("kgram is not of length k"));

  if (kG.find(kgram) != kG.end()) {  // if the kgram is in the map..
  return kG[kgram].at(c);}
  return 0;}

  // (Throw an exception if kgram is not of length k.
  // Throw an exception if no such kgram.)
  char MModel::kRand(std::string kgram) {
  if (kgram.length() != K)
  throw std::invalid_argument("kGram is of different order than K");
  if (kG.find(kgram) == kG.end())
  throw std::invalid_argument("No such kGram");
  // total chance is also
  // int total_chance = kGramFreq[kgram];
  // int total_chance = 0;

  std::vector<int> freqs;
  // std::vector<double> freqqs;
  // acess the list of characters of the kgram. get the total probability
```

```
for (auto it = kG[kgram].begin(); it != kG[kgram].end(); ++it) {
// total_chance += it->second;
freqs.push_back(it->second);
// freqqs.push_back((double) (it->second) / (double) total_chance);
}

// construct a trivial random generator engine from a time-based seed:
unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
std::default_random_engine generator(seed);
std::random_device rd;
std::mt19937 gen(rd());
std::discrete_distribution<int> d(std::begin(freqs), std::end(freqs));

int random_index = d(generator);
char random;
for (auto it = kG[kgram].begin(); it != kG[kgram].end();
--random_index, ++it)  {
if (random_index == 0)
return it->first;
}
// return rand2;
// return random2;
// random =
// std::discrete_distribution<> d({
}

// return a string of length L(text length)
std::string MModel::generate(std::string kgram, int l) {
if (kgram.size() != K)
throw std::invalid_argument("kGram is not of length K");

std::string guessing = kgram;

// new kgram has to be updated, we'll keep old just in case
// std::string kNew = kgram;
char newRand;
for (int i = 0; l > 0; i++ , l--) {
newRand = kRand(kgram);
guessing.push_back(newRand);
kgram.push_back(newRand);
kgram.erase(0, 1);
}
return guessing;
}
```

TextGenerator.cpp

```cpp
int main(int argc, char** argv) {
  int K, L;
  char* temp = argv[1];
  char* temp2 = argv[2];
  std::string inFile = argv[3];
  std::ifstream input;
  input.open(inFile);

  K = atoi(temp);
  L = atoi(temp2);
  std::string s;
  int i;
  std::string total_s;
  while (getline(input, s)) {
  total_s += s;}
  // i++;

  MModel MKov(total_s, K);

  std::cout << MKov.generate("In ", L) << std::endl;
  input.close();
  return 0;
  }
```

PS5: Kronos – Intro to Regular Expression

Objective:

Read in a .log file to evaluate whether a server started. If the server was successfully started, write into a .rpt file and report its duration time. If the server failed, report the failure in the .rpt file.

Utilize Regular Expressions (regex) to compare the input to evaluate whether the input matches the criteria of the regex variable. If it does, create a boost::datetime variable to calculate the uptime of the server.

```
112 (log.c.166) server started: 2013-10-02 18:42:38 Succeeded and ran for 165000ms

855 (log.c.166) server started: 2013-10-03 12:23:21 Succeeded and ran for 174000ms

1568 (log.c.166) server started: 2013-10-04 16:20:03 Succeeded and ran for 183000ms

31956 (log.c.166) server started: 2013-12-03 16:21:13 Succeeded and ran for 175000ms

33032 (log.c.166) server started: 2013-12-04 21:50:27 Succeeded and ran for 150000ms

33390 (log.c.166) server started: 2013-12-04 21:58:45 Succeeded and ran for 149000ms

33920 (log.c.166) server started: 2013-12-04 22:21:03 Succeeded and ran for 148000ms

45538 (log.c.166) server started: 2013-12-05 13:34:25 Succeeded and ran for 150000ms

45858 (log.c.166) server started: 2013-12-05 14:12:25 Succeeded and ran for 148000ms

46353 (log.c.166) server started: 2013-12-05 15:39:02 Succeeded and ran for 147000ms

46600 (log.c.166) server started: 2013-12-05 20:20:24 Succeeded and ran for 150000ms

47035 (log.c.166) server started: 2013-12-10 13:20:43 Succeeded and ran for 149000ms

47991 (log.c.166) server started: 2013-12-10 19:40:58 Succeeded and ran for 177000ms

48341 (log.c.166) server started: 2013-12-11 14:09:11 Succeeded and ran for 150000ms

48636 (log.c.166) server started: 2013-12-11 14:17:49 Succeeded and ran for 177000ms
```

Key algorithms, data structures, or OO designs that were central to this assignment:

Specifically, a regex variable was initialized to match with the criteria of the starting string:

25 string: `" (log.c.166) server started: "`

regex start:`“[(]log.c.[1 - 6][1 - 6][1 - 6][)][\\s]server[\\s]started[\\s]".`

This regex was used as an argument for regex_search:

```
regex_search(getline, smStart, start);
```

Which allowed me to compare the regex with a getline. If the two matched (meaning the getline evaluated as fitting the criteria of the regex) then we would store a substring of the getline to isolate and store the time in a string via

```
startTime = boost::posix_time::time_from_string(starttime);
```

Now with the time stored and the start message received, a regex that fit the criteria of the end string was created and subsequently matched with getline:

```
regex end = "[\\d][\\d][\\d][\\d]-[\\d][\\d]-[\\d][\\d][\\s][\\d][\\d]:[\\d][\\d]:[\\d][\\d].[\\d][\\d][\\d]:INFO:oejs.AbstractConnector:Started[\\s]SelectChannelConnector@[\\d].[\\d].[\\d].[\\d]:[\\d][\\d][\\d][\\d]";
std::regex_search(getline, smEnd, end);
```

We implemented checks by checking if another potential start string was found before the end string, or if there was no end string found. This would indicate the server failed.

What was learned:

PS6 was my first implementation of Regular Expression (regex).

Regex manipulation was used to evaluate criteria of a start and end string.

Boost::datetime then allowed me to manipulate the string into a substring that stored a start time or end time.

```
#include <boost/regex.hpp>
```

```
#include "boost/date_time/gregorian/gregorian.hpp"
#include "boost/date_time/posix_time/posix_time.hpp"

//using namespace boost;
// date get_date
int main(int argc, char** argv)
{
       std::string s;
       std::regex start, end;
       std::string starttime;
       std::string endtime;
       std::smatch smStart;
       std::smatch smEnd;
       start = "[\\d][\\d][\\d][\\d]-[\\d][\\d]-
[\\d][\\d][\\s][\\d][\\d]:[\\d][\\d]:[\\d][\\d]:[\\s][(]log.c.[1 - 6][1 - 6][1 -
6][)][\\s]server[\\s]started[\\s]";
       end = "[\\d][\\d][\\d][\\d]-[\\d][\\d]-
[\\d][\\d][\\s][\\d][\\d]:[\\d][\\d]:[\\d][\\d].[\\d][\\d][\\d]:INFO:oejs.AbstractConnecto
r:Started[\\s]SelectChannelConnector@[\\d].[\\d].[\\d].[\\d]:[\\d][\\d][\\d][\\d]";
       std::ifstream input;
       std::string inFile = argv[1];
       input.open(inFile);
       std::ofstream output;
       std::string outFile = inFile + ".rpt";
       output.open(outFile);
       bool key = 0;
       boost::posix_time::ptime startTime, endTime;
       boost::posix_time::time_duration runTime;

       //keep picking up until you get the server started
              for (int i = 1; getline(input, s); i++) {
                     //this will search for when s and start are the same, and then
increment sm (int) by 1.
                     std::regex_search(s, smStart, start);

                     //if the start gets found..
                     if (smStart.size() > 0) {

                            if (key == 1) {
                            output << i << " (log.c.166) server started: " << starttime <<
" FAILED" << std::endl;
                            }
                            key = 1;

                            //std::cout << "Picked up server start!" << std::endl;
                            //datetime is now the date-time!
                            starttime = s.substr(0, 19);
                            startTime = boost::posix_time::time_from_string(starttime);
                            //std::cout << datetime << std::endl;
                            //advance to the next line
                     }

                     std::regex_search(s, smEnd, end);
                     if (smEnd.size() > 0)
                     {
                            if (key == 0) {
```

```
                            output << i << " (log.c.166) server started: " << starttime <<
" FAILED" << std::endl;
                            continue;
                            }
                            key = 0;
                            //std::cout << "Picked up server end!" << std::endl;
                            endtime = s.substr(0, 19);
                            endTime = boost::posix_time::time_from_string(endtime);
                            //get the time difference
                            runTime = endTime - startTime;
                     output << i  << " (log.c.166) server started: " << starttime << "
Succeeded and ran for " << runTime.total_milliseconds() << "ms" << std::endl;
                     }
                     if (s.back() == EOF) {
                            output << "failed!" << std::endl;
                            continue;
                     }
              }

       input.close();
       output.close();
       return 0;
}
```