PS6: RandWriter

Assignment Overview

This project involved implementing a character-based random text generator using a Markov model of order k. The ‘RandWriter‘ class analyzes a given input string to build a frequency model of all k-length substrings (k-grams) and the probabilities of characters that follow each. The resulting model simulates a probabilistic trajectory through the input text to generate essentially nonsense text that mimics the structure of the original. A client program named ‘TextWriter‘ was developed to take input k and L via command-line arguments, build the model from standard input, and generate a random output string of length L beginning with the first k characters of the text. The higher the order, the closer the text is to the original.

Key Algorithms / Data Structures / OO Designs

The implementation centered around a map-based structure to track all k-grams and their corresponding frequency distributions for the next character. Each k-gram maps to another map of character-to-frequency pairs. The core methods include RandWriter(const string&, size_t), which constructs the circular Markov model, freq(string) and freq(string, char), which return total counts of k-grams and next-character frequencies, kRand(string), which returns a character following a k-gram chosen at random proportionally to its frequency, and generate(string, size_t), which simulates the Markov process to build an output string starting from a seed k-gram. C++’s random utilities (std::mt19937 and std::discrete_distribution) were used to ensure quality seeded randomness. Error checking is implemented for k-gram lengths and undefined k-grams. The project used exception handling and included a series of Boost tests to verify expected behavior and validate robustness.

What I Learned

Through PS6, I learned how to build and apply a Markov model for pseudo-random text generation. I used std::map to build a nested frequency table and efficiently access k-gram data. I leveraged C++ random tools, including std::mt19937 and std::discrete_distribution, for weighted random selection. I also learned to parse and simulate probabilistic models using dynamic data structures. I wrote clean, reusable object-oriented code and client programs for text processing. I constructed robust unit tests with Boost to verify both correct output and exception handling. Lastly, I used lambda functions as parameters in appropriate testing and iteration contexts.

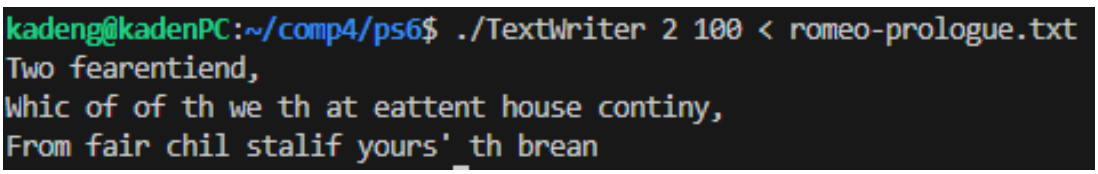

Figure 7: Example Output

Source Code

```
Makefile:
CC = g++
CFLAGS = --std=c++20 -Wall -Werror -pedantic -g
LIB = -lsfml-graphics -lsfml-audio -lsfml-window -lsfml-system -
   lboost_unit_test_framework

DEPS = RandWriter.hpp
SRC = RandWriter.cpp TextWriter.cpp
OBJECTS = RandWriter.o TextWriter.o
TEST_OBJECTS = RandWriter.o test.o

PROGRAM = TextWriter
STATIC_LIB = TextWriter.a
TEST_EXEC = test

.PHONY: all clean lint

all: $(PROGRAM) $(STATIC_LIB) $(TEST_EXEC)

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

```

```
$(PROGRAM): $(OBJECTS)
        $(CC) $(CFLAGS) -o $@ $^ $(LIB)

$(STATIC_LIB): $(OBJECTS)
        ar rcs $@ $^

$(TEST_EXEC): $(TEST_OBJECTS)
        $(CC) $(CFLAGS) -o $@ $^ $(LIB)

clean:
        rm -f *.o $(PROGRAM) $(TEST_EXEC) $(STATIC_LIB)

lint:
        cpplint *.cpp *.hpp

main.cpp (TestWriter.cpp):
// COPYRIGHT 2025 Kaden gardiner
#include "RandWriter.hpp"


int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: ./TextWriter <k> <L>\n";
        return 1;
    }

    size_t k = std::stoi(argv[1]);
    size_t L = std::stoi(argv[2]);

    std::string text((std::istreambuf_iterator<char>(std::cin)), {});
    RandWriter rw(text, k);

    std::string kgram = text.substr(0, k);
    std::string generated = rw.generate(kgram, L);

    std::cout << generated << '\n';
    return 0;
}

RandWriter.hpp:
// COPYRIGHT 2025 Kaden gardiner
#pragma once

#include <string>
#include <stdexcept>
#include <map>
#include <algorithm>
#include <iostream>
#include <random>
#include <chrono>
#include <sstream>
#include <vector>
```

```

class RandWriter {
 public:
    // Create a Markov model of order k from given text
    // Assume that text has length at least k.
    RandWriter(const std::string& str, size_t k);
    std::string getAlphabet() const;
    size_t orderK() const;  // Order k of Markov model
    // Number of occurences of kgram in text
    // Throw an exception if kgram is not length k
    int freq(const std::string& kgram) const;
    // Number of times that character c follows kgram
    // if order=0, return num of times that char c appears
    // (throw an exception if kgram is not of length k)
    int freq(const std::string& kgram, char c) const;
    // Random character following given kgram
    // (throw an exception if kgram is not of length k)
    // (throw an exception if no such kgram)
    char kRand(const std::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
    std::string generate(const std::string& kgram, size_t l);

 private:
    std::string _text;
    size_t _k;
    std::map<std::string, std::map<char, int>> _kgramMap;
    mutable std::mt19937 _engine;
    std::vector<char> _alphabet;
};

RandWriter.cpp:
// COPYRIGHT 2025 Kaden gardiner
#include "RandWriter.hpp"

RandWriter::RandWriter(const std::string& str, size_t k) :
    _text(str), _k(k) {
    if (str.length() < k) {
        throw std::logic_error("Invalid length");
    }
    _engine.seed(std::chrono::steady_clock::now().time_since_epoch().count
       ());
    std::string circular = _text + _text.substr(0, _k);
    for (size_t i = 0; i < _text.length(); ++i) {
        std::string kgram = circular.substr(i, _k);
        char next = circular[i + _k];
        _kgramMap[kgram][next]++;
        if (std::find_if(_alphabet.begin(), _alphabet.end(),
```

```
                        [&](char ch) { return ch == next; }) == _alphabet.
                           end()) {
            _alphabet.push_back(next);
        }
    }
}

std::string RandWriter::getAlphabet() const {
    return std::string(_alphabet.begin(), _alphabet.end());
}

size_t RandWriter::orderK() const {
    return _k;
}

int RandWriter::freq(const std::string& kgram) const {
    if (kgram.length() != _k) {
        throw std::logic_error("kgram must be at least the length of k");
    }
    auto it = _kgramMap.find(kgram);
    if (it == _kgramMap.end()) {
        std::cout << "kgram not found" << std::endl;
        return 0;
    }
    int total = 0;
    for (const auto& ent : it->second) {
        const auto& count = ent.second;
        total += count;
    }
    return total;
}


int RandWriter::freq(const std::string& kgram, char c) const {
    if (kgram.length() != _k) {
        throw std::logic_error("kgram must be at least the length of k");
    }
    std::map<std::string, std::map<char,
    int>>::const_iterator it = _kgramMap.find(kgram);
    if (it == _kgramMap.end()) {
        std::cout << "kgram not found" << std::endl;
        return 0;
    }
    std::map<char, int>::const_iterator inner = it->second.find(c);
    if (inner == it->second.end()) {
        return 0;
    }
    return inner->second;
}

char RandWriter::kRand(const std::string& kgram) {
    if (kgram.length() < _k) {
```

```
        throw std::logic_error("kgram must be at least the length of k");
    }
    int total = freq(kgram);
    if (total == 0) {
        throw std::logic_error("No valid transitions for kgram");
    }

    std::uniform_int_distribution<> dist(1, total);
    int r = dist(_engine);
    int sum = 0;

    for (char c : _alphabet) {
        int f = freq(kgram, c);
        if (f == 0) continue;
        sum += f;
        if (r <= sum) return c;
    }
    throw std::runtime_error("Random selection failed");
}

std::string RandWriter::generate(const std::string& kgram, size_t l) {
    if (kgram.length() < _k) {
        throw std::logic_error("kgram must be at least length of k");
    }
    if (l < _k) {
        throw std::logic_error("L must be at least k");
    }
    std::string result = kgram;
    std::string current = kgram;
    while (result.length() < l) {
        char next = kRand(current);
        result += next;
        current = result.substr(result.length() - _k);
    }
    return result;
}

test.cpp:
// COPYRIGHT 2025 Kaden gardiner
#define BOOST_TEST_MODULE EDistanceTest
#include <boost/test/included/unit_test.hpp>
#include "RandWriter.hpp"

BOOST_AUTO_TEST_CASE(kZeroTest) {
    RandWriter rw("gagggagaggcgagaaa", 0);
    BOOST_REQUIRE_THROW(rw.generate("g", 10), std::logic_error);
}

BOOST_AUTO_TEST_CASE(wrongLengthTest) {
    RandWriter rw("gagggagaggcgagaaa", 2);
    std::string kgram = "ga";
    size_t L = 100;
```

```
    std::string output = rw.generate(kgram, L);

    BOOST_REQUIRE_EQUAL(output.length(), L);
    BOOST_REQUIRE_EQUAL(output.substr(0, kgram.length()), kgram);
}

BOOST_AUTO_TEST_CASE(wrongDistributionTest) {
    RandWriter rw("gagggagaggcgagaaa", 0);

    std::map<char, int> counts;
    for (int i = 0; i < 10000; ++i) {
        char c = rw.kRand("");
        counts[c]++;
    }

    double g_ratio = counts['g'] / 10000.0;
    double a_ratio = counts['a'] / 10000.0;
    double c_ratio = counts['c'] / 10000.0;

    BOOST_REQUIRE(g_ratio > 0.45 && g_ratio < 0.55);
    BOOST_REQUIRE(a_ratio > 0.35 && a_ratio < 0.45);
    BOOST_REQUIRE(c_ratio > 0.03 && c_ratio < 0.08);
}
```