```
 1: CC= g++
 2: CFLAGS= -Wall -Werror -ansi -pedantic --std=c++11
 3: #SFMLFLAGS= -lsfml-system
 4: Boost= -lboost_unit_test_framework
 5: all: test markov
 6: markov: TextGenerator.o markov.o
 7:         $(CC)  TextGenerator.o markov.o  -o markov
 8: test: test.cpp markov.o
 9:         $(CC) test.cpp markov.o $(CFLAGS) $(Boost) -o test
10: TextGenerator.o: TextGenerator.cpp markov.h
11:         $(CC) -c TextGenerator.cpp markov.h
12: markov.o: markov.cpp  markov.h
13:         $(CC) -c markov.cpp markov.h  $(CFLAGS)
14: clean:
15:         rm *.o
16:         rm markov
17:         rm test
18:
```

```
    1: // "Copyright 2020 <Greg Kaplowitz>"
    2: #include "markov.h"
    3:
    4: int main(int argc, char *argv[]) {
    5:   int k;
    6:   int L;
    7:   k = atof(argv[1]);
    8:   L = atof(argv[2]);  // get cmd line args
    9:   std::cout << k << "       " << L << std::endl;
   10:
   11:   std::string input = "";
   12:   std::string current_txt = "";
   13:
   14:   while (std::cin >> current_txt) {
   15:   input += " " + current_txt;
   16:   current_txt = "";
   17:   }
   18:   std::cout << "The story:" << std::endl << std::endl;
   19:   for (int a = 0; a < L; a++) {  // print the passing texted
   20:   std::cout << input[a];
   21:
   22:
   23:   if (input[a] == '.' || input[a] == '!') {
   24:   std::cout << std::endl;  // auto enters at end of statement so I dont have
 to
   25:   }
   26:   }
   27:
   28:   std::string output_string = "";
   29:   MarkovModel amazing(input, k);
   30:   output_string += "" + amazing.generate(input.substr(0, k), L);
   31:   std::cout << std::endl;
   32:   std::cout << "The fanfic:" << std::endl;
   33:
   34:   for (int a = 0; a < L; a++) {
   35:     std::cout << output_string[a];
   36:
   37:     if (output_string[a] == '.' || output_string[a] == '!') {
   38:       std::cout << std::endl;  // auto enters at end of statement so I
   39:   // don't have to
   40:     }
   41:   }
   42:
   43:
   44:
   45:   return 0;
   46: }
   47:
```

```
    1: // "Copyright 2020 <Greg Kaplowitz>"
    2:
    3: #ifndef MARKOV_H
    4: #define MARKOV_H
    5:
    6: #include <map>
    7: #include <iostream>
    8: #include <stdexcept>
    9: #include <algorithm>
   10: #include <string>
   11:
   12: class MarkovModel {
   13:  public:
   14:   MarkovModel(std::string text, int k);
   15:   int k_order();
   16:   int freq(std::string _kgram);
   17:   int freq(std::string _kgram, char c);
   18:   char kRand(std::string _kgram);
   19:   std::string generate(std::string _kgram, int L);
   20:
   21:   friend std::ostream& operator<< (std::ostream &out, MarkovModel &mm);
   22:
   23:  private:
   24:   std::string txt;
   25:   int order;
   26:   std::map <std::string, int> kgram;
   27:   std::string symtab;
   28: };
   29:
   30: int countFreq(std::string &pat, std::string &txt);  //NOLINT
   31: int count_following_freq(std::string &pat, std::string &txt, char c);  //NOL
INT
   32:
   33: #endif
   34:
```

```
    1: // "Copyright 2020 <Greg Kaplowitz>"
    2:
    3: #include "markov.h"
    4: #include <vector>
    5: #include <utility>
    6:
    7: MarkovModel::MarkovModel(std::string text, int k) {
    8:   order = k;  // set order
    9:   // make text circular
   10:   for (int i = 0; i < order-1; i++) {
   11:   text.push_back(text[i]);
   12:   }
   13:   std::cout << text.length() << std::endl;
   14:   symtab.push_back(text.at(0));
   15:   for (unsigned int i = 0; i < text.length(); i++) {
   16:   bool present = false;
   17:
   18:   for (unsigned int j=0; j < symtab.length(); j++) {  // checks if the curre
nt
   19:   //  char of the text is already in or symbol table
   20:   // std::cout<<symtab.length()<<std::endl;
   21:   if (symtab.at(j) == text.at(i)) {  // if yes
   22:   present = true;  // then say so
   23:   }
   24:   }
   25:   if (!present) {  // if it isnt present then add it
   26:   symtab.push_back(text.at(i));
   27:   }
   28:   }
   29:   // we should have a filled out sybol table here
   30:   std::cout << symtab << std::endl;
   31:   for (unsigned int i=0; i < text.length()-1; i++) {  // fill out the kgram
and
   32:   // freq sections of markov chart
   33:   std::string kgram_string = text.substr(i, order);  // gets the kgram
   34:   int freq = countFreq(kgram_string, text);  // gets the freq
   35:   kgram.insert(std::pair<std::string, int>(kgram_string, freq));
   36:   // adds the kgram key with its freq data
   37:   }
   38:   // now we should have a filled out map representing the kgram and freqency
   39:   // for each
   40:   /*for (std::map<std::string, int>::iterator it = kgram.begin();it !=kgram.
end();it++){
   41:   std::cout<< it->first <<"  "<<it->second<<std::endl;// test
   42:   }*/ //this prints out the map
   43:   txt = text;
   44:   }
   45:
   46:
   47: int MarkovModel::k_order() {
   48:         return order;
   49: }
   50:
   51: int MarkovModel::freq(std::string _kgram) {
   52:   if (_kgram.length() != (unsigned)order){
   53:   throw
   54:   std::invalid_argument("kgram not of length k 1");
   55:   }
   56:   if (kgram.find(_kgram) == kgram.end()) {
   57:   return 0;
   58:   }
```

```
   59:   std::map<std::string, int > ::iterator it;
   60:   it = kgram.find(_kgram);
   61:   return it->second;
   62: }
   63:
   64: int MarkovModel::freq(std::string _kgram, char c) {
   65:   if (_kgram.length() != (unsigned) order) {
   66:   throw
   67:   std::invalid_argument("kgram not of length k 2");
   68:   }
   69:   if (kgram.find(_kgram) == kgram.end()) {
   70:   return 0;
   71:   }
   72:   return count_following_freq(_kgram, txt, c);
   73: }
   74:
   75: char MarkovModel::kRand(std::string _kgram) {
   76:   if (_kgram.length() != (unsigned) order) {
   77:   throw
   78:   std::invalid_argument("kgram not of length k 3");
   79:   }
   80:   char result;
   81:   int freq = this->freq(_kgram);
   82:   int RNG = rand() % freq + 1;
   83:   //std::cout<<RNG;
   84:   bool is_zero = false;
   85: //      The way I implement the weighted random sallection here starts a ran
dom
   86: //      integer between
   87: //      1 and the frequency of the kgram then subtract the character freqenc
y
   88: //      so if there are 3 symbols in the symtab a/c/g with 5 kgram frequency

   89: //      we get a random int of 3 and the char freq's are a=3 c=0 g=2
   90: //      we subtract 3 from the freq and check to see if the running count is
 <=0
   91: //      we are at 0 so a is returned
   92:   for (unsigned int i =0; i<symtab.length(); i++) {  // iterate through the
   93:   // symtab
   94:   int diff = this->freq(_kgram, symtab[i]);
   95:   RNG -= diff;
   96:   if (RNG<=0 && !is_zero) {
   97:   result = symtab[i];
   98:   is_zero = true;
   99:   }
  100:   }
  101:   return result;
  102:  }
  103:
  104:   std::string MarkovModel::generate(std::string _kgram, int L) {
  105:   if (_kgram.length() != (unsigned) order){
  106:   throw
  107:   std::invalid_argument("kgram not of length k 4");
  108:   }
  109:   std::string txt = "";
  110:   txt += "" + _kgram;
  111:
  112:   for (int i=0; i<L-order;i++) {
  113:   char x = this->kRand(txt.substr(i, order));
  114:   // std::cout<<x<<std::endl;
  115:   txt.push_back(x);
```

```
  116:   }
  117:   return txt;
  118:   }
  119:
  120:   std::ostream& operator<< (std::ostream &out, MarkovModel &markov) {
  121:   out<<"First the map: "<<std::endl;
  122:   for (std::map<std::string, int>::iterator it = markov.kgram.begin();it !=m
arkov.kgram.end();it++) {// no lint
  123:   out<< it->first <<"  "<<it->second<<std::endl;// print the map;
  124:   }
  125:
  126:   out << "The symtab: "<< std::endl;
  127:   out << markov.symtab << std::endl;
  128:
  129:   out << "The order: "<<std::endl;
  130:   out << markov.order << std::endl;
  131:
  132:   return out;
  133: }
  134:
  135: int countFreq(std::string &pat, std::string &txt) {
  136:   int M = pat.length();
  137:   int N = txt.length();
  138:   int res = 0;
  139:
  140:   /* A loop to slide pat[ ] one by one */
  141:   for (int i = 0; i <= N - M; i++) {
  142:   /* For current index i, check for
  143:   pattern match */
  144:   int j;
  145:   for (j = 0; j < M; j++)
  146:   if (txt[i+j] != pat[j])
  147:   break;
  148:
  149:   // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
  150:   if (j == M) {
  151:   res++;
  152:   j = 0;
  153:   }
  154:   }
  155:   return res;
  156: }
  157:
  158: int count_following_freq(std::string &pat, std::string &txt, char c) {//NOLI
NT
  159:   int M = pat.length();
  160:   int N = txt.length();
  161:   int res = 0;
  162:   /* A loop to slide pat[ ] one by one */
  163:   for (int i = 0; i <= N - M; i++) {
  164:   /* For current index i, check for
  165:   pattern match */
  166:   int j;
  167:   for (j = 0; j < M; j++)
  168:   if (txt[i+j] != pat[j])
  169:   break;
  170:   // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
  171:   if (j == M) {
  172:   if (txt[i + j] == c) {
  173:   res++;
  174:   }
```

```
  175:   j = 0;
  176:   }
  177:   }
  178:   return res;
  179: }
```

```
 1: // "Copyright 2020 <Greg Kaplowitz>"
 2: #define BOOST_TEST_DYN_LINK
 3: #define BOOST_TEST_MODULE Main
 4: #include <boost/test/unit_test.hpp>
 5: #include <iostream>
 6: #include <string>
 7: #include "markov.h"
 8:
 9: BOOST_AUTO_TEST_CASE(kgram_freq) {
10:   MarkovModel m1("gagggagaggcgagaaa", 2);
11:   BOOST_REQUIRE(m1.freq("aa") == 2);
12: }
13:
14: BOOST_AUTO_TEST_CASE(char_freq) {
15:   MarkovModel m1("gagggagaggcgagaaa", 2);
16:   BOOST_REQUIRE( m1.freq("aa", 'a') == 1);
17:   BOOST_REQUIRE( m1.freq("aa", 'c') == 0);
18:   BOOST_REQUIRE( m1.freq("aa", 'g') == 1);
19: }
20:
21: BOOST_AUTO_TEST_CASE(freq_without_a_valid_kgram) {
22:   MarkovModel m1("gagggagaggcgagaaa", 2);
23:    BOOST_REQUIRE(m1.freq("az") == 0);
24:     BOOST_REQUIRE( m1.freq("aa", 'z') == 0);
25: }
26:
27: BOOST_AUTO_TEST_CASE(invalid_argument) {
28:   MarkovModel m1("gagggagaggcgagaaa", 2);
29:   BOOST_REQUIRE_THROW(m1.freq("aaaa"), std::invalid_argument);
30:   BOOST_REQUIRE_THROW(m1.freq("aaaa", 'a'), std::invalid_argument);
31:    BOOST_REQUIRE_THROW(m1.kRand("aaaa"), std::invalid_argument);
32:    BOOST_REQUIRE_THROW(m1.generate("aaaa", 3), std::invalid_argument);
33: }
34:
35: BOOST_AUTO_TEST_CASE(generate) {
36:   MarkovModel m1("gagggagaggcgagaaa", 2);
37:    BOOST_REQUIRE(m1.generate("gg", 10) == "ggagagaggg");
38: }
39:
40:
```