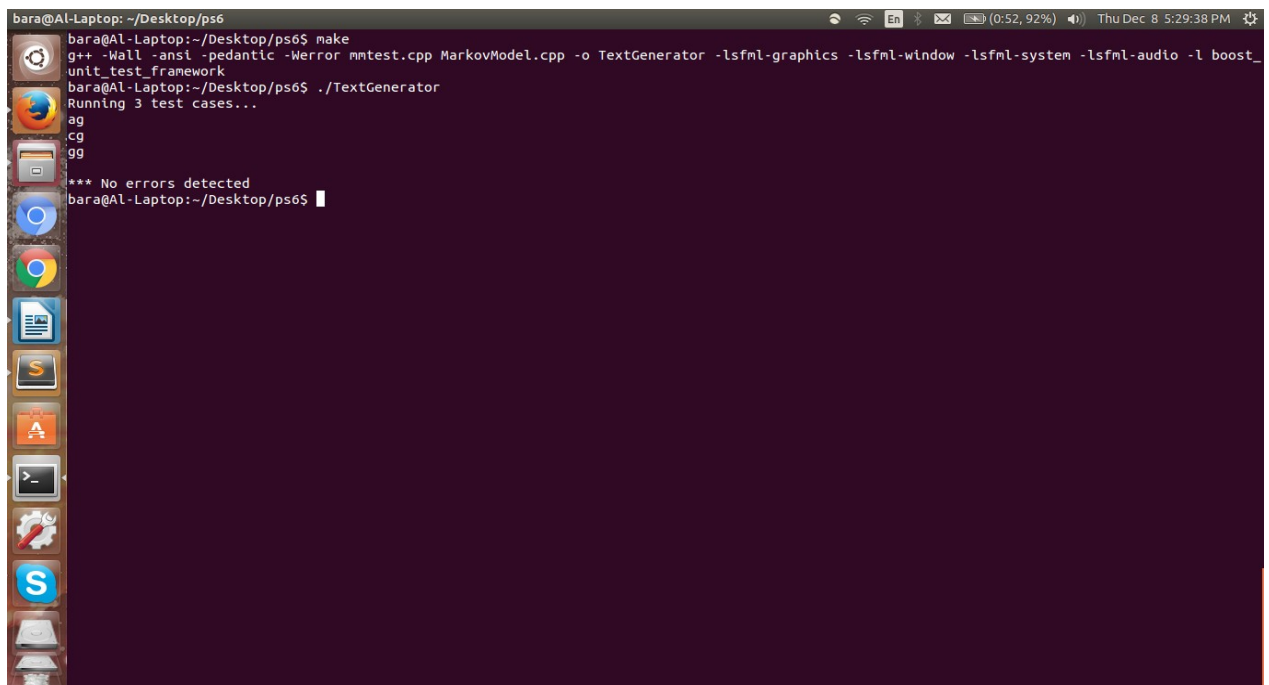


## PS6: Markov Model of Natural Language

The focus on this assignment to convert the concept of the Markov Model into code. The Markov model is concept that predicts each letter for the next and finds the probability of what will be the next set of letters. To accomplish this assignment we needed to create a Markov model of order k that are given from a given text of type string. Then using a Markov chain to get the current state in a k-gram of the next character from the selected random probability from Markov model.

One key library we used was the map library. The map function is a container that can store elements formed by a combination of key values an mapped values in any specific order. In the constructor, I saved the string and number of k values to the member variables. Then created a string that took each character and pushed back the letter. I needed to check the char if its already in the alphabet by having a for loop and iterator through the matrix. Then insert the character into the map. In the randk function, I would have a random value from the kgram and then finally print it out. The gen function, would print out the next kgram until the size of string . Then the operator would be called each time the map needed to be printed out.

Learning about the data type map using to store different data types. I understood how the Markov model could be used in different projects. Thankfully, I had help from a tutor to help explain the lay out of this program and what is needed.



```
bara@Al-Laptop: ~/Desktop/ps6
bara@Al-Laptop:~/Desktop/ps6$ make
g++ -Wall -ansi -pedantic -Werror mmtest.cpp MarkovModel.cpp -o TextGenerator -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio -l boost_
unit_test_framework
bara@Al-Laptop:~/Desktop/ps6$ ./TextGenerator
Running 3 test cases...
ag
cg
gg
*** No errors detected
bara@Al-Laptop:~/Desktop/ps6$
```

```
1: cc = g++
2:
3: all : MarkovModel
4:
5: MarkovModel : MarkovModel.o mmttest.o
6:      $(cc) -Wall -ansi -pedantic -Werror mmttest.cpp MarkovModel.cpp -o TextGenerator -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio -l boost_unit_test_framework
7:
8: MarkovModel.o : MarkovModel.hpp
9:      $(cc) -c MarkovModel.cpp -o MarkovModel.o
10:
11: mmttest.o : MarkovModel.hpp
12:      $(cc) -c mmttest.cpp -o mmttest.o
13:
14: clean :
15:      rm *.o TextGenerator debug
16:
17: debug :
18:      g++ -g -Wall -ansi -pedantic -Werror mmttest.cpp MarkovModel.cpp -o Markov -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio -l boost_unit_test_framework
```

```
1: /*<Copyright Fred Martain*/
2:
3: #define BOOST_TEST_DYN_LINK
4: #define BOOST_TEST_MODULE Main
5: #include <boost/test/unit_test.hpp>
6:
7: #include <iostream>
8: #include <string>
9: #include <exception>
10: #include <stdexcept>
11:
12: #include "MarkovModel.hpp"
13:
14: // using namespace std;
15:
16: BOOST_AUTO_TEST_CASE(order0) {
17:     // normal constructor
18:     BOOST_REQUIRE_NO_THROW(MarkovModel("gagggagagggcgagaaa", 0));
19:
20:     MarkovModel mm("gagggagagggcgagaaa", 0);
21:
22:     BOOST_REQUIRE(mm.order() == 0);
23:     // length of input in constructor
24:     BOOST_REQUIRE(mm.freq("") == 17);
25:     BOOST_REQUIRE_THROW(mm.freq("x"), std::runtime_error);
26:
27:     BOOST_REQUIRE(mm.freq("", 'g') == 9);
28:     BOOST_REQUIRE(mm.freq("", 'a') == 7);
29:     BOOST_REQUIRE(mm.freq("", 'c') == 1);
30:     BOOST_REQUIRE(mm.freq("", 'x') == 0);
31: }
32:
33: BOOST_AUTO_TEST_CASE(order1) {
34:     // normal constructor
35:     BOOST_REQUIRE_NO_THROW(MarkovModel("gagggagagggcgagaaa", 1));
36:
37:     MarkovModel mm("gagggagagggcgagaaa", 1);
38:
39:     BOOST_REQUIRE(mm.order() == 1);
40:     BOOST_REQUIRE_THROW(mm.freq(""), std::runtime_error);
41:     BOOST_REQUIRE_THROW(mm.freq("xx"), std::runtime_error);
42:
43:     BOOST_REQUIRE(mm.freq("a") == 7);
44:     BOOST_REQUIRE(mm.freq("g") == 9);
45:     BOOST_REQUIRE(mm.freq("c") == 1);
46:
47:     BOOST_REQUIRE(mm.freq("a", 'a') == 2);
48:     BOOST_REQUIRE(mm.freq("a", 'c') == 0);
49:     BOOST_REQUIRE(mm.freq("a", 'g') == 5);
50:
51:     BOOST_REQUIRE(mm.freq("c", 'a') == 0);
52:     BOOST_REQUIRE(mm.freq("c", 'c') == 0);
53:     BOOST_REQUIRE(mm.freq("c", 'g') == 1);
54:
55:     BOOST_REQUIRE(mm.freq("g", 'a') == 5);
56:     BOOST_REQUIRE(mm.freq("g", 'c') == 1);
57:     BOOST_REQUIRE(mm.freq("g", 'g') == 3);
58:
59:     BOOST_REQUIRE_NO_THROW(mm.randk("a"));
60:     BOOST_REQUIRE_NO_THROW(mm.randk("c"));
61:     BOOST_REQUIRE_NO_THROW(mm.randk("g"));
```

```
62:
63:     BOOST_REQUIRE_THROW(mm.randk("x"), std::runtime_error);
64:
65:     BOOST_REQUIRE_THROW(mm.randk("xx"), std::runtime_error);
66: }
67:
68: BOOST_AUTO_TEST_CASE(order2) {
69:     // normal constructor
70:     BOOST_REQUIRE_NO_THROW(MarkovModel("gagggagagggcgagaaa", 2));
71:
72:     MarkovModel mm("gagggagagggcgagaaa", 2);
73:
74:     BOOST_REQUIRE(mm.order() == 2);
75:
76:     BOOST_REQUIRE_THROW(mm.freq(""), std::runtime_error);
77:     BOOST_REQUIRE_THROW(mm.freq("x"), std::runtime_error);
78:     BOOST_REQUIRE_NO_THROW(mm.freq("xx"));
79:     // kgram is wrong length
80:     BOOST_REQUIRE_THROW(mm.freq("", 'g'), std::runtime_error);
81:     // kgram is wrong length
82:     BOOST_REQUIRE_THROW(mm.freq("x", 'g'), std::runtime_error);
83:     // kgram is wrong length
84:     BOOST_REQUIRE_THROW(mm.freq("xxx", 'g'), std::runtime_error);
85:
86:
87:     BOOST_REQUIRE(mm.freq("aa") == 2);
88:     BOOST_REQUIRE(mm.freq("aa", 'a') == 1);
89:     BOOST_REQUIRE(mm.freq("aa", 'c') == 0);
90:     BOOST_REQUIRE(mm.freq("aa", 'g') == 1);
91:
92:     BOOST_REQUIRE(mm.freq("ag") == 5);
93:     BOOST_REQUIRE(mm.freq("ag", 'a') == 3);
94:     BOOST_REQUIRE(mm.freq("ag", 'c') == 0);
95:     BOOST_REQUIRE(mm.freq("ag", 'g') == 2);
96:
97:     BOOST_REQUIRE(mm.freq("cg") == 1);
98:     BOOST_REQUIRE(mm.freq("cg", 'a') == 1);
99:     BOOST_REQUIRE(mm.freq("cg", 'c') == 0);
100:    BOOST_REQUIRE(mm.freq("cg", 'g') == 0);
101:
102:    BOOST_REQUIRE(mm.freq("ga") == 5);
103:    BOOST_REQUIRE(mm.freq("ga", 'a') == 1);
104:    BOOST_REQUIRE(mm.freq("ga", 'c') == 0);
105:    BOOST_REQUIRE(mm.freq("ga", 'g') == 4);
106:
107:    BOOST_REQUIRE(mm.freq("gc") == 1);
108:    BOOST_REQUIRE(mm.freq("gc", 'a') == 0);
109:    BOOST_REQUIRE(mm.freq("gc", 'c') == 0);
110:    BOOST_REQUIRE(mm.freq("gc", 'g') == 1);
111:
112:    BOOST_REQUIRE(mm.freq("gg") == 3);
113:    BOOST_REQUIRE(mm.freq("gg", 'a') == 1);
114:    BOOST_REQUIRE(mm.freq("gg", 'c') == 1);
115:    BOOST_REQUIRE(mm.freq("gg", 'g') == 1);
116: }
```

```
1: /* <Copyright Abara Mehene*/
2: #ifndef MARKOV_MODEL_HPP
3: #define MARKOV_MODEL_HPP
4:
5: #include <iostream>
6: #include <map>
7: #include <string>
8: #include <stdexcept>
9: #include <algorithm>
10:
11: class MarkovModel {
12: private:
13:     int _order;
14:     std::map<std::string, int> _kgrams; // must #include <map>
15:     std::string _alphabet;
16:     // space
17: public:
18:     // create a Markov model of order k from given text
19:     // Assume that text has length at least k.
20:     MarkovModel(std::string text, int k);
21:
22:     // order k of Markov model
23:     int order();
24:
25:     // number of occurrences of kgram in text
26:     // (throw an exception if kgram is not of length k)
27:     int freq(std::string kgram);
28:
29:     // number of times that character c follows kgram
30:     // if order=0, return num of times char c appears
31:     // (throw an exception if kgram is not of length k)
32:     int freq(std::string kgram, char c);
33:
34:     // random character following given kgram
35:     // (Throw an exception if kgram is not of length k.
36:     // Throw an exception if no such kgram.)
37:     char randk(std::string kgram);
38:
39:     // generate a string of length T characters
40:     // by simulating a trajectory through the corresponding
41:     // Markov chain. The first k characters of the newly
42:     // generated string should be the argument kgram.
43:     // Throw an exception if kgram is not of length k.
44:     // Assume that T is at least k.
45:     std::string gen(std::string kgram, int T);
46:
47:     // overload the stream insertion operator and display
48:     // the internal state of the Markov Model. Print out
49:     // the order, the alphabet, and the frequencies of
50:     // the k-grams and k+1-grams.
51:     friend std::ostream& operator<<(std::ostream &out, MarkovModel &mm);
52:
53:     ~MarkovModel();
54: };
55:
56: #endif
```

```
1: /* <Copyright Abara Mehene*/
2:
3: #include "MarkovModel.hpp"
4: #include <utility>
5: #include <map>
6: #include <string>
7:
8: MarkovModel::MarkovModel(std::string text, int k) {
9:     _order = k;
10:    _alphabet = text;
11:
12:    char character;
13:    bool value = 0;
14:    std::string circularString = text;
15:    std::string tempString;
16:
17:    // INSERTING CHARS INTO THE ALPHABET
18:
19:    // store the chars that appear in text into the alphabet
20:    for (int i = 0; i < _order; ++i) {
21:        // insert the new char into the alphabet
22:        circularString.push_back(text[i]);
23:    }
24:
25:    // check if the char in the text is already in the alphabet
26:    for (unsigned int i = 0; i < text.length(); ++i) {
27:        character = text.at(i);
28:        value = 0;
29:        // check if we already have the char
30:        for (unsigned int j = 0; j < _alphabet.length(); ++j) {
31:            if (_alphabet.at(j) == character)
32:                value = 1;
33:        }
34:        // do we store the char into the alphabet?
35:        if (!value)
36:            _alphabet.push_back(character);
37:    }
38:
39:
40:    std::map<std::string, int>::iterator it;
41:    int temp_count = 0;
42:
43:    // get a substring from the text and inserting it into kgram
44:    for (int x = _order; x <= _order + 1; ++x) {
45:        for (unsigned int y = 0; y < text.length(); ++y) {
46:            tempString = circularString.substr(y, x);
47:            _kgrams.insert(std::pair<std::string, int>(tempString, 0));
48:            it = _kgrams.find(tempString);
49:            temp_count = it->second;
50:            temp_count++;
51:            _kgrams[tempString] = temp_count;
52:        }
53:    }
54: }
55:
56: // returns order
57: int MarkovModel::order() {
58:     return _order;
59: }
60:
61: int MarkovModel::freq(std::string kgram) {
```

```
62: // error check
63: if ((unsigned)_order != kgram.length())
64:     throw std::runtime_error("Kgram is not of length k");
65: // space
66: std::map<std::string, int>::iterator numOfkGram;
67: // go through the map and count how many
68: // times we have the string kgram
69: numOfkGram = _kgrams.find(kgram);
70:
71: // return the kgram we find
72: if (numOfkGram == _kgrams.end())
73:     return 0;
74: return numOfkGram->second;
75: }
76:
77: int MarkovModel::freq(std::string kgram, char c) {
78:     // error check
79:     if (kgram.length() != (unsigned)_order)
80:         throw std::runtime_error("Kgram is not of length k");
81:
82:     // put c into kgram then find the new kgram
83:     std::map<std::string, int>::iterator numOfkGram;
84:     kgram.push_back(c);
85:     numOfkGram = _kgrams.find(kgram);
86:
87:     // if there is the kgram
88:     if (numOfkGram == _kgrams.end())
89:         return 0;
90:     return numOfkGram->second;
91: }
92:
93: char MarkovModel::randk(std::string kgram) {
94:     unsigned int seed = time(NULL);
95:     int randomValue;
96:     std::string randomInput;
97:
98:     // error check
99:     if (kgram.length() != (unsigned)_order)
100:         throw std::runtime_error("Kgram is not of length k");
101:     // space
102:     // try to find the kgram
103:     std::map<std::string, int>::iterator temp;
104:     temp = _kgrams.find(kgram);
105:
106:     int kgram_freq = freq(kgram);
107:
108:     // if there is no such kgram
109:     if (temp == _kgrams.end())
110:         throw std::runtime_error("No such kgram");
111:
112:     for (;;) {
113:         // gets random value from the kgram_freq
114:         randomValue = rand_r(&seed) % kgram_freq;
115:         randomInput = kgram + _alphabet[randomValue];
116:         // if we are at the end return a random char
117:         if (temp != _kgrams.end()) {
118:             std::cout << randomInput << std::endl;
119:             return _alphabet[randomValue];
120:         }
121:     }
122: }
```

```
123:
124: std::string MarkovModel::gen(std::string kgram, int T) {
125:     // error check
126:     if (kgram.length() != (unsigned)_order)
127:         throw std::runtime_error("Kgram is not of length k");
128:
129:     // put the initial kgram into our string
130:     std::string tempkGram = kgram;
131:
132:     // append the random character to the end of our output
133:     // until size the string is of size T
134:     for (int i = kgram.length(); i < T; ++i) {
135:         tempkGram.push_back(randk(kgram));
136:     }
137:
138:     return tempkGram;
139: }
140:
141: std::ostream& operator<<(std::ostream &out, MarkovModel &mm) {
142:     std::map<std::string, int>::iterator it;
143:     // basic output
144:     out << "Order: " << mm._order << std::endl;
145:     out << "Alphabet: " << mm._alphabet << std::endl;
146:     out << "Kgrams map: " << std::endl;
147:
148:     for (it == mm._kgrams.begin(); it != mm._kgrams.end(); it++) {
149:         out << it->first << it->second << std::endl;
150:     }
151:     return out;
152: }
153:
154: MarkovModel::~MarkovModel() {
155:     // destructor
156: }
```