Contents

1 Basic Test Results

```
Running...
Opening tar file
SpamDetector.cpp
HashMap.hpp
OK
Tar extracted O.K.
Checking files...
OK
Making sure files are not empty...
OK


*************************************************************************
 IMPORTANT MESSAGE:
 This presubmission test has 2! Parts.
 Each has own compilation process,
 so keep in mind to check both parts are compiled successfully.
 if first part won't compile the test will NOT proceed to second part.
*************************************************************************



============================== PART 1 ================================

Compilation check...

Compiling...
OK

Compiling...
OK
Compilation seems OK! Check if you got warnings!


=========================================================================
 Public test cases
 Please Note:
 You must pass at least one test in order to not fail the presubmit
=========================================================================


=============================================================================================
IMPORTANT NOTICE:
  This presubmission script is NOT testing your program exit code so check it manually
  The exit codes will be checked while grading your submission only
=============================================================================================

====================
Running test...
OK
Running test...
OK
Test OS Succeeded.
Info: db and SPAM email
====================

====================
Running test...
OK
```

```
Running test...
OK
Test ON Succeeded.
Info: db and not spam email
====================

====================
Running test...
OK
Running test...
OK
Test 6I Succeeded.
Info: Invalid Input test
====================

====================
Running test...
OK
Running test...
OK
Test 6I Succeeded.
Info: Invalid Input test
====================

============================ PART 1 - END ==============================


============================ PART 2 ====================================

Compilation check...

Compiling...
OK

Compiling...
OK

Compilation seems OK! Check if you got warnings!
Running test...
[#0][Presubmission] Test __presubmit_testCreateHashMaps... OK!
[#1][Presubmission] Test __presubmit_testInsert... OK!
[#2][Presubmission] Test __presubmit_testSize... OK!
[#3][Presubmission] Test __presubmit_testCapacity... OK!
[#4][Presubmission] Test __presubmit_testEmpty... OK!
[#5][Presubmission] Test __presubmit_testClear... OK!
[#6][Presubmission] Test __presubmit_testBucketSize... OK!
[#7][Presubmission] Test __presubmit_testGetElement... OK!
[#8][Presubmission] Test __presubmit_testContainsKey... OK!
[#9][Presubmission] Test __presubmit_testAssignment... OK!
[#10][Presubmission] Test __presubmit_testComparison... OK!
[#11][Presubmission] Test __presubmit_testIterator... OK!
[#12][Presubmission] Test __presubmit_testVectorsCtor... OK!
[#13][Presubmission] Test __presubmit_testCopyCtor... OK!
OK

Check output above!
============================ PART 2 - END ==============================



========================
= Checking coding style =
========================
 ** Total Violated Rules      : 0
 ** Total Errors Occurs       : 0
 ** Total Violated Files Count: 0
```

2 HashMap.hpp

```cpp
//
// Created by brahan on 23/01/2020.
//

#ifndef CPP_EX3_HASHMAP_HPP
#define CPP_EX3_HASHMAP_HPP

#include <vector>
#include <stdexcept>

using std::vector;
#define INVALID "Invalid input"
#define LOWER_THRESHOLD 0.25
#define UPPER_THRESHOLD 0.75
#define BASIC_TABLE_SIZE 16
#define EMPTY 0
#define INVALID_IDX -1
#define TO_DELETE 0
#define TO_ADD 1
#define MIN_CAPACITY 1
#define RESIZE_FACTOR 2

//class MemoryAllocationException : public std::exception //TODO we dont need it?
//{
//    const char *what() const noexcept override
//    {
//        return "Memory not allocated";
//    }
//} MemoryAllocationException;

/**
 * template for the hash map
 * @tparam KeyT defines the keys in the hashmap
 * @tparam ValueT defines the values in teh hashmap
 */
template<typename KeyT, typename ValueT>
class HashMap
{
    typedef std::pair<KeyT, ValueT> cell;
    typedef std::vector<std::pair<KeyT, ValueT>> bucket;

public:
    /**
     * inner class that represent a iterator
     */
    class iterator
    {
    public:
        // iterator typedef, needed for iterator after cpp14
        typedef cell value_type;

        typedef cell *pointer;

        typedef cell &reference;

        typedef int difference_type;

        typedef std::forward_iterator_tag iterator_category;

```

```
60          /**
61           * default constructor
62           */
63          iterator() = default;
64
65          /**
66           * default destructor
67           */
68          ~iterator() = default;
69
70
71          /**
72           * iterator that gets map
73           * @param map
74           */
75          explicit iterator(const HashMap<KeyT, ValueT> *map, int curBucket = 0) : _map(map), _curBucket(curBucket)
76          {
77              if (map != nullptr)
78              {
79                  curIdx = 0;
80                  if (curIdx == (int) _map->_table[_curBucket].size())
81                  {
82                      _current = nullptr;
83                      operator++();
84                  }
85                  _current = &(_map->_table[_curBucket][curIdx]);
86              }
87              else
88              {
89                  curIdx = 0;
90                  _current = nullptr;
91              }
92
93          }
94
95          //operators: //TODO iterator lecture slides 15
96          reference operator*() const
97          {
98              return *_current;
99          }
100
101         pointer operator->() const
102         {
103             return _current;
104         }
105
106         /**
107          * ++ after
108          * @return
109          */
110         iterator &operator++()
111         {
112             ++curIdx;
113             while (curIdx > (((int) _map->_table[_curBucket].size()) - 1) && _curBucket < ((int) _map->capacity() - 1))
114             {
115                 curIdx = 0;
116                 _curBucket++;
117             }
118             if (_curBucket == ((int) _map->capacity() - 1))
119             {
120                 _current = nullptr;
121                 return (*this);
122             }
123             _current = &(_map->_table[_curBucket][curIdx]);
124             return (*this);
125         }
126
127         /**
```

```
128             * ++ before
129             * @return this after the increment
130             */
131            iterator operator++(int)
132            {
133                iterator tmp = *this;
134                operator++();
135                return tmp;
136            } //tODO done
137
138            /**
139             * comparison operator
140             * @param other other iterator
141             * @return true if the the current cell of both iterators are equal
142             */
143            bool operator==(const iterator &other) const
144            {
145                return _current == other._current;
146
147            }//TODO done
148            /**
149             * comparison operator
150             * @param other other iterator
151             * @return true if the the current cell of both iterators are not equal
152             */
153            bool operator!=(const iterator &other) const
154            {
155                return !(other == *this);
156            } //TODO done
157
158            /**
159             * assignment operator for the iterator
160             * @param other the other iterator
161             * @return the iterator after the assignment
162             */
163            iterator &operator=(const iterator &other)
164            {
165                this->map = *other.map;
166                this->bucketIdx = other.bucketIdx;
167                this->_current = other._current;
168                return *this;
169            }
170
171        private:
172            const HashMap<KeyT, ValueT> *_map;
173            int _curBucket, curIdx;
174            cell *_current;
175        };
176
177    private:
178        bucket *_table;
179        size_t _size, _capacity;
180
181        /**
182         *  calculate the key hash value== its bucket
183         * @param key the key
184         * @return the key hash value
185         */
186        int _hash(const KeyT &key) const
187        {
188            return (std::hash<KeyT>{}(key) & (_capacity - 1));
189        }
190
191        /**
192         * the function with search for a key and return its index in the map
193         * @param key the key
194         * @return the key place inside its bucket
195         */
```

```
196     int _containsHelper(const KeyT &key) const
197     {
198         int bucketIdx = _hash(key);
199         for (int i = 0; i < (int) _table[bucketIdx].size(); ++i)
200         { //iterate once on all the bucket --> O(n)
201             if (_table[bucketIdx].at(i).first == key)
202             {
203                 return i;
204             }
205         }
206         return INVALID_IDX;
207     }
208 
209     /**
210      * helper for constructor, operators =\[] insert without checking if the key is inside the table
211      * @param k key
212      * @param v value
213      */
214     void _insertWithDupes(KeyT k, ValueT v)
215     {
216         ++_size;
217         int idx = _containsHelper(k);
218         int bucketIdx = (std::hash<KeyT>{}(k) & (capacity() - 1));
219         if (idx == INVALID_IDX)
220         {
221             std::pair<KeyT, ValueT> nP(k, v);
222             _table[bucketIdx].push_back(nP);
223         }
224         else
225         {
226             _table[bucketIdx][idx].second = v;
227         }
228     }
229 
230     /**
231      * adds items from other map with dupes
232      * @param old old hashmap
233      */
234     void _addingLoop(HashMap<KeyT, ValueT> &old)
235     {
236         for (int i = 0; i < old.capacity(); ++i)
237         {
238             if (!old._table[i].empty())
239             {
240                 for (auto &pair:old._table[i])
241                 {
242                     _insertWithDupes(pair.first, pair.second);
243                 }
244             }
245         }
246         old.clear();
247     }
248 
249     /**
250      * resizs the hashmap
251      * @param rehashBigger 1 if we need to resize to a bigger hashmap, 0 otherwize
252      */
253     void _resize(int rehashBigger)
254     {
255         double modifiedLoad = 0;
256         if (rehashBigger == TO_ADD)
257         {
258             modifiedLoad = (double) (_size + 1) / (double) _capacity;
259             if (UPPER_THRESHOLD < modifiedLoad)
260             {
261                 HashMap<KeyT, ValueT> old(*this);
262                 _capacity *= RESIZE_FACTOR;
263                 _size = 0;
```

```
                delete[] _table;
                _table = new bucket[_capacity];
                _addingLoop(old);
            }
        }
        else if (rehashBigger == TO_DELETE)
        {
            modifiedLoad = (double) (_size) / (double) _capacity;
            if (LOWER_THRESHOLD > modifiedLoad && _capacity > MIN_CAPACITY)
            {
                HashMap<KeyT, ValueT> old(*this);
                _capacity /= RESIZE_FACTOR;
                _size = 0;
                delete[] _table;
                _table = new bucket[_capacity];
                _addingLoop(old);
            }
        }
    }

public:

    /**
     * init empty hashmap
     */
    HashMap() : _table(new bucket[BASIC_TABLE_SIZE]), _size(EMPTY), _capacity(BASIC_TABLE_SIZE)
    {

    }

    /**
     * init hashmap from vectors and keys
     * @param keys keys vector
     * @param values values vector
     */
    HashMap(vector<KeyT> keys, vector<ValueT> values) : _size(EMPTY), _capacity(BASIC_TABLE_SIZE)
    {
        if (keys.size() != values.size()) // ||keys.capacity()!=values.capacity()
        {
            throw std::invalid_argument("constructor");
        }
        _table = new std::vector<std::pair<KeyT, ValueT>>[_capacity];
        for (int i = 0; i < (int) keys.size(); ++i)
        {
            KeyT k = keys[i];
            ValueT v = values[i];
            _resize(TO_ADD);
            int place = _containsHelper(k);
            int bucketIdx = (std::hash<KeyT>{}(k) & (_capacity - 1));
            if (place == INVALID_IDX)
            {
                std::pair<KeyT, ValueT> nP(k, v);
                _table[bucketIdx].push_back(nP);
                ++_size;
            }
            else
            {
                _table[bucketIdx][place].second = v;
//                ++_size;
            }
        }
    }

    /**
     * copy constructor
     * @param other the other hashmap
     */
    HashMap(HashMap<KeyT, ValueT> &other) : _size(other._size), _capacity(other._capacity)
```

```
    {
        _table = new bucket[other._capacity];
        for (int i = 0; i < (int) other._capacity; ++i)
        {
            _table[i] = other._table[i];
        }
    }

    /**
     * destructor
     */
    ~HashMap()
    {
        delete[]_table;
    }

    /**
     * getter for the number of element in the hashmap
     * @return the hashmap size
     */
    int size() const
    {
        return (int) _size;
    }

    /**
     * getter for the table capacity
     * @return the hashmap capacity
     */
    int capacity() const
    {
        return (int) _capacity;
    }

    /**
     * return true if the hashmap is empty
     * @return true if empty, false otherwise
     */
    bool empty() const
    {
        return (_size == 0);
    }

    /**
     * insert a pair into the hashmap
     * @return true if the value have been succsefuly inserted , false otherwise
     */
    bool insert(const KeyT key, const ValueT value)
    {
        if (_containsHelper(key) == INVALID_IDX)
        {
            _resize(TO_ADD);
            int place = (std::hash<KeyT>{}(key) & (capacity() - 1));
            std::pair<KeyT, ValueT> nP(key, value);
            _table[place].push_back(nP);
            ++_size;
            return true;
        }
        return false;
    }

    /**
     * checks if a key is inside the table
     * @param key the key we want to search for
     * @return true if the key is in the table, false otherwise
     */
    bool containsKey(KeyT key) const
    {
```

```
400             return (_containsHelper(key) != INVALID_IDX); // On
401         }
402 
403         /**
404          * returns the value of the given key in the table
405          * @param key the key we want its value
406          * @return the key value
407          */
408         ValueT &at(KeyT key) const
409         {
410             int bucketIdx = _hash(key);
411             int keyIdx = _containsHelper(key); //On
412             if (keyIdx != INVALID_IDX)
413             {
414                 return _table[bucketIdx][keyIdx].second;
415             }
416             else
417             {
418                 throw std::invalid_argument("the key not in map - at()");
419             }
420         }
421 
422         /**
423          * delete the key value from the table
424          * @param key the key
425          * @return true if the value successfully deleted, false otherwise
426          */
427         bool erase(KeyT key)
428         {
429             int bucketIdx = _hash(key);
430             int keyIdx = _containsHelper(key); //On
431             if (keyIdx != INVALID_IDX)
432             {
433                 for (typename std::vector<std::pair<KeyT, ValueT>>::const_iterator it = _table[bucketIdx].begin();
434                      it != _table[bucketIdx].end(); it++)
435                 {
436                     if (it->first == key)
437                     {
438                         _table[bucketIdx].erase(it);
439                         --_size;
440                         _resize(TO_DELETE);
441                         return true;
442                     }
443                 }
444             }
445             return false;
446         }
447 
448         /**
449          * return the table load factor
450          * @return the table for the load factor
451          */
452         double getLoadFactor() const
453         {
454             return (double) _size / (double) _capacity;
455         }
456 
457         /**
458          * gets a key and returns his bucket size
459          * @param key the key
460          * @return the key bucket size
461          */
462         int bucketSize(KeyT key) const
463         {
464             int keyIdx = _containsHelper(key); //On
465             if (keyIdx != INVALID_IDX)
466             {
467                 int bucketIdx = _hash(key);
```

```
468             return _table[bucketIdx].size();
469         }
470         else
471         {
472             throw std::invalid_argument("the key not in map - bucketSize()");
473         }
474     }
475 
476     /**
477      * gets a key and returns his bucket index
478      * @param key the key
479      * @return the key bucket index
480      */
481     int bucketIndex(KeyT key) const
482     {
483         int keyIdx = _containsHelper(key); //On
484         if (keyIdx != INVALID_IDX)
485         {
486             int bucketIdx = _hash(key);
487             return bucketIdx;
488         }
489         else
490         {
491             throw std::invalid_argument("the key not in map - at()");
492         }
493     }
494 
495     /**
496      * delete all the map items without changing the capacity
497      */
498     void clear()
499     {
500         for (int i = 0; i < this->capacity(); ++i)
501         {
502             _table[i].clear();
503         }
504         _size = 0;
505     }
506 /// operators:
507     /**
508      * random access operator for the map
509      * @param k the key we want to get its value
510      * @return the key value
511      */
512     ValueT &operator[](const KeyT &k) noexcept
513     {
514         int bucketIdx = (std::hash<KeyT>{}(k) & (capacity() - 1));
515         int place = _containsHelper(k);
516         if (place == INVALID_IDX)
517         {
518             _resize(TO_ADD);
519             ValueT v = ValueT();
520             _insertWithDupes(k, v);
521             place = _containsHelper(k);
522             bucketIdx = (std::hash<KeyT>{}(k) & (capacity() - 1));
523         }
524         return _table[bucketIdx][place].second;
525 
526     }
527 
528     /**
529      * random access operator for the map
530      * @param k the key we want to get its value
531      * @return the key value
532      */
533     ValueT &operator[](const KeyT &k) const noexcept
534     {
535         int place = _containsHelper(k);
```

```
536             int bucketIdx = (std::hash<KeyT>{}(k) & (capacity() - 1));
537             if (place == INVALID_IDX)
538             {
539                 //DO None
540             }
541             return _table[bucketIdx][place].second;
542         }
543 
544         /**
545          * comparison operator for the map
546          * @param other the other map
547          * @return true if the maps are equals
548          */
549         bool operator==(const HashMap<KeyT, ValueT> &other) const
550         {
551             if (this->size() != other.size())
552             {
553                 return false;
554             }
555             for (int i = 0; i < capacity(); ++i)
556             {
557                 for (int j = 0; j < (int) _table[i].size(); ++j)
558                 {
559                     if (_table[i].at(j) != other._table[i].at(j))
560                     {
561                         return false;
562                     }
563                 }
564             }
565             return true;
566         }
567 
568         /**
569          * comparison operator for the map
570          * @param other the other map
571          * @return true if the maps are equals
572          */
573         bool operator!=(const HashMap<KeyT, ValueT> &other) const
574         {
575             return !(*this == other);
576         }
577 
578         /**
579          * assignment operator for the map
580          * @param other other map
581          * @return the map after the assignment
582          */
583         HashMap<KeyT, ValueT> &operator=(const HashMap<KeyT, ValueT> &other)
584         {
585             this->_size = other._size;
586             this->_capacity = other._capacity;
587             _table = new std::vector<std::pair<KeyT, ValueT>>[_capacity];
588             for (int i = 0; i < _capacity; ++i)
589             {
590                 _table[i] = other._table[i];
591             }
592             return *this;
593         }
594 
595         /**
596          * begin
597          * @return iterator that point to the first item of the map
598          */
599         const iterator begin() const
600         {
601             return (iterator(this, 0));
602         }
603 
```

```
604         /**
605          * const version of begin
606          * @return iterator that point to the first item of the map
607          */
608         iterator cbegin() const
609         {
610             return begin();
611         }
612 
613         /**
614          * end
615          * @return iterator that point to the end of the map
616          */
617         iterator end() const
618         {
619             return (iterator(nullptr));
620         }
621 
622         /**
623          * const version of end
624          * @return iterator that point to the end of the map
625          */
626         iterator cend() const
627         {
628             return end();
629         }
630     };
631 
632     #endif //CPP_EX3_HASHMAP_HPP
```

3 SpamDetector.cpp

```cpp
#include <iostream>
#include <vector>
#include <fstream>
#include <regex>
#include <boost/tokenizer.hpp>
#include <unordered_map>
#include <random>
#include "HashMap.hpp"

// input handlers:
#define USAGE "Usage: SpamDetector <database path> <message path> <threshold>"
#define NUM_ARGS 4
// input idx:
#define FILE_IDX 1
#define FIRST_COL 0
#define SECOND_COL 1
#define MSG_IDX 2
#define THRESHOLD_IDX 3
// msg outputs:
#define SPAM "SPAM"
#define NOT_SPAM "NOT_SPAM"
#define COMMA ","
#define LINE_PATTERN  R"(^((.*)\,)?\d+[\r\n]?$)"
//#define CSV_FILE ".csv" //TODO we dont need to check if its ends with csv
//#define FILE_SUFF_LEN 4
#define NON_NEGATIVE 0

/**
 * handles error msg and return values
 * @return value that indicate that the progarm has failed
 */
int errMsg()
{
    std::cerr << INVALID << std::endl;
    return EXIT_FAILURE;
}

/**
 * checks if a given string is digit
 * @param num
 * @return
 */
bool checkIfDigit(std::string num)
{
    for (int i = 0; i < (int) num.length(); ++i)
    {
        if (!std::isdigit(num[i]))
        {
            return false;
        }
    }
    return true;
}

/**
 * count how many time a substring is in the string
 * @param txt out txt
 * @param pat out pattern
 * @return the number of time the pattern is in the text
```

```
 */
int subStringCounter(const std::string &txt, const std::string &pat)
{
    int txtLen = txt.length();
    int patLen = pat.length();
    int counter = 0;
    for (int i = 0; i <= txtLen - patLen; i++)
    {
        int j;
        for (j = 0; j < patLen; j++)
        {
            if (txt[i + j] != pat[j])
            {
                break;
            }
        }
        if (j == patLen)
        {
            counter++;
            j = 0;
        }
    }
    return counter;
}

/**
 * helper method that validate the line in the file
 * @param line a single file line
 * @param p the line regex
 * @return 1 if there is error, 0 else
 */
int _validateLine(const std::string &line)
{
    if (line.empty())
    {
        return EXIT_FAILURE;
    }
    else if (subStringCounter(line, ",") != 1)
    {
        return EXIT_FAILURE;
    }
    int startIdx = line.find(',');
    std::string curNum = line.substr(startIdx + 1, line.length() - 1);
    if (curNum.empty() || !checkIfDigit(curNum))
    {
        return EXIT_FAILURE;
    }
    int number = std::stoi(curNum);
    if (number < NON_NEGATIVE)
    {
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}

/**
 * parse the file data
 * @param path the file path
 * @param isValid indicator if the file is valid
 * @return vector with the file data if everything ok
 */
std::vector<std::vector<std::string> > parseFile(const std::string &path, int &isValid)
{
    std::ifstream file(path);
    if (file.bad() || file.fail())
    {
        file.close();
        isValid = EXIT_FAILURE;
```

```
128     }
129     boost::char_separator<char> separator{COMMA};
130     std::vector<std::vector<std::string> > dataList;
131     std::string line;
132     while (getline(file, line))
133     {
134         if (_validateLine(line) == EXIT_FAILURE)
135         {
136             file.close();
137             isValid = EXIT_FAILURE;
138             break;
139         }
140         boost::tokenizer<boost::char_separator<char>> pattern{line, separator};
141         std::vector<std::string> parsedLine;
142         for (const auto &t:pattern)
143         {
144             parsedLine.push_back(t);
145         }
146         dataList.push_back(parsedLine);
147     }
148     file.close();
149     return dataList;
150 }
151 
152 /**
153  * create the vectors for teh HashMap
154  * @param data the parsed csv file
155  * @param firstCol the keys
156  * @param secondCol the values
157  */
158 void createVectors(vector<vector<std::string>> data, vector<std::string> &firstCol, vector<int> &secondCol)
159 {
160     for (int j = (int) data.size() - 1; j >= 0; --j)
161     {
162         firstCol.push_back(data.at(j).at(FIRST_COL));
163         secondCol.push_back(stoi(data.at(j).at(SECOND_COL)));
164     }
165 }
166 
167 /**
168  * iterate over a string and make its letters lowercase
169  * @param data the data we want to change to lower case
170  */
171 void makeLowerCase(vector<std::string> &data)
172 {
173     for (int j = (int) data.size() - 1; j >= 0; --j)
174     {
175         for (int i = 0; i < (int) data.at(j).length(); ++i)
176         {
177             data.at(j)[i] = tolower(data.at(j)[i]);
178         }
179     }
180 }
181 
182 /**
183  * parses the text file
184  * @param path the txt file path
185  * @param isValid a flag to check if the data is valid
186  * @return parsed txt file
187  */
188 vector<std::string> parseTxt(const std::string &path, int &isValid)
189 {
190     std::ifstream file(path);
191     if (file.bad() || file.fail())
192     {
193         file.close();
194         isValid = EXIT_FAILURE;
195     }
```

```
196     boost::char_separator<char> separator{COMMA};
197     std::vector<std::string> dataList;
198     std::string line;
199     while (getline(file, line))
200     {
201         dataList.push_back(line);
202     }
203     file.close();
204     return dataList;
205 }
206 
207 /**
208  * print the spam \ notspam according to the thresholdand the spamvalue
209  * @param th our threshold
210  * @param spamValue the spamvalue of the txt msg
211  */
212 void checkForThreshold(int th, int spamValue)
213 {
214     if (spamValue >= th)
215     {
216         std::cout << SPAM << std::endl;
217     }
218     else
219     {
220         std::cout << NOT_SPAM << std::endl;
221     }
222 }
223 
224 /**
225  * check the value of all the spam words in teh txt file
226  * @param database the spam words database
227  * @param txtData the txt data
228  * @param words the spam words data
229  * @return the spam value in the txt file
230  */
231 int countValue(const HashMap<std::string, int> &database, vector<std::string> &txtData, vector<std::string> &words)
232 {
233     int spamValue = 0;
234     int counter = 0;
235     for (std::string &s:txtData)
236     {
237         for (std::string &word:words)
238         {
239             counter = subStringCounter(s, word);
240             spamValue += counter * database.at(word);
241         }
242     }
243     return spamValue;
244 }
245 
246 /**
247  * main function
248  * @param argc number of arguments
249  * @param argv program arguments
250  * @return 1 succ,0 fail
251  */
252 int main(int argc, const char *argv[])
253 {
254     if (argc != NUM_ARGS)
255     {
256         std::cerr << USAGE << std::endl;
257         return EXIT_FAILURE;
258     }
259     int isValid = 0;
260     vector<vector<std::string> > csv = parseFile(argv[FILE_IDX], isValid);
261     if (isValid)
262     {
263         return errMsg();
```

```
264         }
265         vector<std::string> words;
266         vector<int> points;
267         createVectors(csv, words, points);
268         makeLowerCase(words);
269         try
270         {
271             const HashMap<std::string, int> database(words, points);
272             vector<std::string> txtData = parseTxt(argv[MSG_IDX], isValid);
273             if (isValid)
274             {
275                 return errMsg();
276             }
277             makeLowerCase(txtData);
278             int spamValue = countValue(database, txtData, words);
279             std::string threshold = argv[THRESHOLD_IDX];
280             if (!checkIfDigit(threshold))
281             {
282                 return errMsg();
283             }
284             int thres = std::stoi(threshold);
285             if (thres <= NON_NEGATIVE)
286             {
287                 return errMsg();
288             }
289             checkForThreshold(thres, spamValue);
290             return EXIT_SUCCESS;
291         }
292         catch (int e)
293         {
294             std::cerr << INVALID << std::endl;
295             return EXIT_FAILURE;
296         }
297     }
```